UNIVERSIDAD NACIONAL DE COLOMBIA

Desarrollo de un programa de computador para el análisis lineal de estructuras aporticadas tridimensionales sometidas a cargas estáticas

Cristian Danilo Ramirez Vargas

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería Civil y Agrícola
Bogotá D. C., Colombia
2021

Desarrollo de un programa de computador para el análisis lineal de estructuras aporticadas tridimensionales sometidas a cargas estáticas

Cristian Danilo Ramirez Vargas

Tesis presentada como requisito parcial para optar al título de:
Magister en Estructuras

Director(a):
Ph. D. Martín Estrada Mejia

Línea de Investigación:
Análisis de estructuras
Grupo de Investigación:
Análisis, diseño y materiales - GIES

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería Civil y Agrícola
Bogotá D. C., Colombia
2021

Hombres inteligentes gran pensantes
Hemos creado un monstruo
Las bombas radiactivas y nucleares
Que descompondrán la humanidad
Quien totalmente se autodestruirá

Ya creador no hay para volver a comenzar
Como dijo la sagrada maldición
El universo en siete días lo creó

Razas de todos los colores
Tomemos una reacción
Potencias monopolizadoras
Analicen esta situación
Países tercermundistas
De brazos no nos crucemos
Acabemos pronto con esto

Futuro nunca habrá
Futuro nunca ha habido
Este en mundo que esta perdido
Dependiendo de un botón
Y de la decisión
De un idealista cabrón

La tercera guerra mundial
Será un estruendo nuclear
Donde historiadores no podrán narrarla
Y los humanos no podremos resistirla

Las invenciones científicas
Lejos de liberar de la ignorancia
Y del trabajo envilecedor
Lo aumentan
Y hacen más refinada la servidumbre

—*La ciencia de la autodestrucción*, La Pestilencia (1989)

Agradecimientos

No habría podido hacer este trabajo sin la dirección del profesor Martín Estrada. Su conocimiento del mundo de la programación me ayudó en momentos decisivos durante el desarrollo del código. Gracias a él trabajé con la librería *Three.js*. No sé como hacer para agradecerle por su paciencia.

Este trabajo también se debe al curso *Computación Visual* del profesor Jean Pierre Charalambos. Su descripción del *grafo* y como trabajar con la *escena* fue lo que me permitió hacer *FEM.js*.

Adicionalmente, apliqué el concepto de *cuaternión* en el problema de la rotación de los ejes de referencia tiempo después de haberlo estudiado en una de sus clases, lo que me permitió implementar el método de análisis matricial de manera innovadora. Gracias a su curso ahora creo entender muchas cosas que de adolescente siempre quise saber.

También quiero agradecer a la profesora Maritzabel Molina ya que mi entendimiento del método de análisis matricial proviene de su curso de *análisis estructural básico*. A ella nos debemos muchos ingenieros estructurales.

Así mismo, quiero agradecer al profesor Fernando Ramírez, de la Universidad de los Andes, por enseñarme el *método de los elementos finitos*, y al profesor Dorian Linero por enseñarme a implementarlo. A ellos gracias por haberme permitido ganar confianza con el método.

Finalmente, quiero agradecer la ayuda de la Coordinación Curricular del Posgrado en Estructuras, especialmente a la profesora Caori Takeuchi quien no tuvo reparos en dejarme ver el curso de Computación Visual. Ese día comenzó la verdadera tesis.

Contenido

Lista de Figuras

Lista de Algoritmos

1 Introducción

Los programas de computador comerciales para el análisis y diseño de estructuras que se encuentran vigentes a la fecha cuentan, en general, con un entorno gráfico que le permite al usuario describir el modelo de forma interactiva, procesarlo y visualizar los resultados de manera conveniente.

En Escamilla, 1995 se presenta una lista de algunos de estos programas de uso común en América Latina, entre los cuales se encuentra *ETABS* (Three Dimensional Analysis of Building Systems - Extended Version).

ETABS es un programa de computador creado por Edward Wilson, Jeffery Hollings y Henry Dovey en 1975. Según Wilson et al, 1975, este programa de computador fue desarrollado para el análisis estructural lineal de edificios de pórticos y muros a cortante sujetos tanto a cargas estáticas como sísmicas. El edificio es idealizado como un sistema de elementos aporticados y muros a cortante independientes interconectado por losas de entrepiso las cuales son rígidas en su propio plano.

Este programa es una extensión de *TABS* (Three Dimensional Analysis of Building Systems) para poder analizar pórticos tridimensionales. Según Wilson y Dovey, 1972, una de las razones para desarrollar TABS fue darle una retroalimentación a los usuarios de los programas *FRMSTC* (Static Load Analysis of High-Rise Buildings), *FRMDYN* (Dynamic Analysis of Multistory Buildings), *LATERAL* y *SOLID SAP* (Static Analysis Program for Three-Dimensional Solid Structures).

FRMSTC permitía analizar edificios simétricos con pórticos y muros a cortante paralelos sujetos a cargas estáticas y evaluar los modos y las frecuencias. FRMDYN era similar a FRMSTC con la excepción que la carga era la aceleración del terreno debido a un desplazamiento dependiente del tiempo. LATERAL fue una extensión de FRMSTC que permitía analizar linealmente pórticos y muros a cortante que no eran necesariamente paralelos con tres grados de libertad en cada piso. SOLID SAP era un programa general de elementos finitos y tenía una opción que permitía introducir la aproximación de piso rígido. Este programa también tenía la opción de realizar análisis dinámico.

En la actualidad, ETABS se encuentra en la versión 18.1.1 y según Computers y Structures, 2020, puede ser ejecutado en computadores con sistema operativo Windows 7, Windows 8 o Windows 10 con arquitectura de 64 bits que cuenten como mínimo con un procesador Intel Pentium 4 o AMD Athlon 64, una resolución de 1024x768 pixeles con 16 bits por canal, 8 GB de RAM y 6 GB de espacio en el disco duro. En la figura **1-1** se presenta la ventana del programa ETABS ejecutándose en un computador con Windows 10.

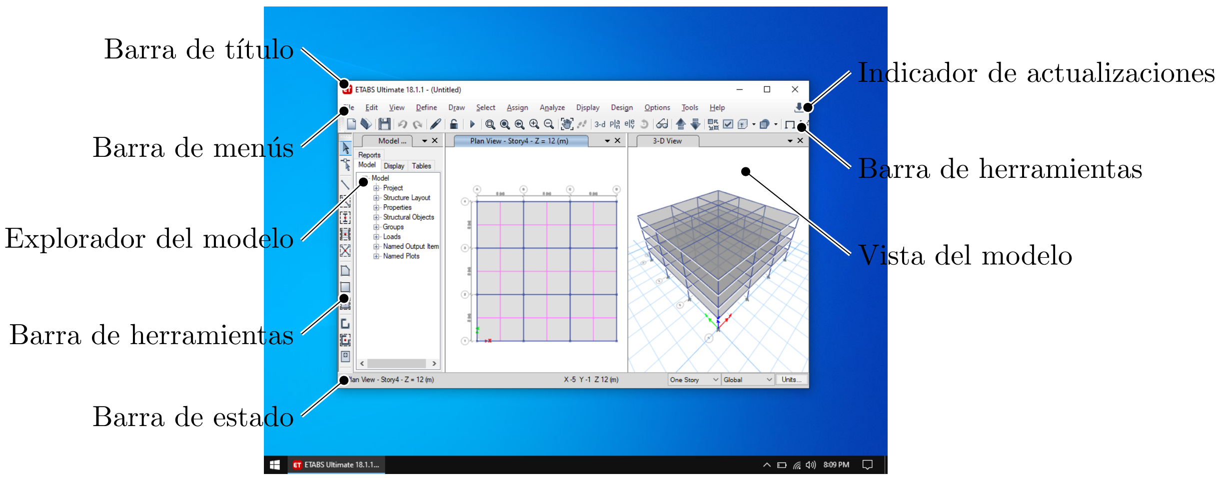

Figura 1-1: Ventana del programa ETABS ejecutandose en Windows 10.

A través de múltiples cuadros de diálogo, los cuales son accesibles ya sea a través de la barra de menús, las barras de herramientas, el explorador del modelo, las vistas del modelo o con atajos de teclado, el usuario es capaz de modelar la estructura que desea analizar al describir los materiales, las secciones transversales, los elementos estructurales, las condiciones de apoyo, los diafragmas y las cargas.

Según Computers y Structures, 2017, ETABS analiza el modelo usando el motor de análisis *SAPFire*, el cual es común a otros programas de la misma compañia (*SAP2000*, *SAFE* y *CSiBridge*). SAPFire es la última versión de la serie de programas *SAP* y ofrece las siguientes herramientas:

- Análisis estático y dinámico,
- Análisis lineal y no lineal,
- Análisis sísmico y análisis incremental no lineal (*pushover*),
- Análisis de cargas móviles,
- No linealidad geométrica, incluyendo efectos P-delta y grandes desplazamientos,
- Etapas constructivas,

- Fluencia lenta (*creep*), retracción (*shrinkage*) y envejecimiento,
- Análisis de pandeo,
- Análisis de densidad espectral de potencia y estado estacionario,
- Elementos aporticados y laminares, incluyendo el comportamiento de vigas, columnas, cerchas, membranas y placas,
- Elementos tipo cable y tendón,
- Elementos bidimensionales planos y elementos sólidos asimétricos,
- Elementos sólidos tridimensionales,
- Resortes no lineales y apoyos,
- Propiedades de los resortes y apoyos dependientes de la frecuencia,

Con los resultados del análisis del modelo, el posprocesador de ETABS puede *diseñar* los elementos estructurales de acuerdo a uno de varios códigos de diseño de diferentes países. ETABS es capaz de diseñar pórticos en acero, pórticos en concreto, vigas compuestas, columnas compuestas, vigas en acero de alma abierta (*steel joist*), muros a cortante y losas de concreto.

Adicionalmente, según Computers y Structures, 2019, ETABS cuenta con la posibilidad de generar dibujos estructurales esquemáticos de las plantas estructurales, de los despieces de vigas, columnas y muros a cortante, y de los detalles de las conexiones de acero.

En términos generales, estos programas de computador comerciales cuenta con características similares a las de ETABS. Actualmente, dichos programas están innovando para permitirle al usuario trabajar con modelos *BIM* (Building Information Modeling).

1.1. Objetivo

Desarrollar un programa de computador a código abierto para el análisis lineal de estructuras aporticadas tridimensionales sometidas a cargas estáticas.

Con este trabajo se pretende contribuir al ejercicio libre de la profesión del ingeniero estructural y a la enseñanza del análisis de las estructuras.

1.1.1. Objetivos específicos

- Desarrollar el módulo de análisis estructural para calcular el desplazamiento de los nudos, el valor de las reacciones y de las fuerzas internas de los elementos de una estructura sometida a cargas estáticas.

- Desarrollar el ambiente gráfico y la interfaz gráfica de usuario del programa de computador para permitirle al usuario ingresar los datos que describen la estructura, las acciones a las cuales se encuentra sometida y visualizar los resultados del análisis estructural.

1.2. Metodología

Se desarrollaron los programas de computador *pyFEM* y *FEM.js*. El primero para analizar estructuras aporticadas tridimensionales sometidas a cargas estáticas y el segundo para modelarlas. Esto con el fin que FEM.js pueda ser usado junto con otro programa de computador diferente a pyFEM.

Durante el desarrollo de estos programas, así como el de este documento, se utilizó *git* como sistema de control de versiones. Según Chacon, 2014, git es un sistema distribuido de control de versiones que registra los cambios realizados a un conjunto de archivos para coordinar el trabajo entre programadores.

Una copia de los repositorios de pyFEM y FEM.js se encuentran en la página de internet *GitHub*, la cual permite alojar proyectos utilizando git. pyFEM está alojado en https://github.com/rvcristiand/pyFEM mientras que FEM.js está alojado en https://github.com/rvcristiand/FEM.js.

1.2.1. pyFEM

pyFEM fue desarrollado en *Python*. Según Lutz, 2013, Python es un lenguaje de programación interpretado orientado a objetos cuya filosofía hace enfasis en la legibilidad de su código. Los archivos revelantes que componen el repositorio de pyFEM son:

```
pyFEM/
├── LICENSE
├── README.md
├── example_1.json
├── example_2.json
├── example_3.json
├── pyFEM/
│   ├── classtools.py
│   ├── core.py
│   └── primitives.py
└── test/
    ├── space_frame.py
    └── trusses.py
```

El archivo `LICENCE` contiene la licencia de pyFEM mientras que el archivo `README.md` contiene todas las instrucciones necesarias para ejecutar y usar pyFEM.

La carpeta *pyFEM* contiene los archivos `classtools.py`, `core.py` y `primitives.py`, los cuales contienen las instrucciones para analizar los modelos. La extensión *py* se usa para indicar que los archivos son programas de Python.

En el archivo `classtools.py` se encuentran las *clases* `UniqueInstances` y `AttrDisplay`, la primera para evitar que se creen *objetos* de una misma clase con la misma información y la segunda para generar una representación conveniente de los objetos.

En el archivo `primitives.py` se encuentran varias clases, entre ellas `Material`, `Section`, `Joint`, `Frame`, `Support`, `LoadPattern`, etc., las cuales permiten describir los diferentes atributos del modelo.

En el archivo `core.py` se encuentra la clase `Structure` la cual permite describir estructuras para ser analizados. Para crear objetos de esta clase se debe llamar la clase indicando los grados de libertad a tener en cuenta en el análisis. A partir de un objeto de esta clase es posible describir el modelo de la estructura al agregar materiales, secciones transversales, nodos, elemenetos aporticados, apoyos, patrones de carga, cargas en los nodos y cargas distribuidas en los elementos aporticados.

Los archivos `example_1.json`, `example_2.json` y `example_3.json` almacenan los modelos de tres de los ejemplos presentados en Escamilla, 1995 que han sido analizados con pyFEM. La extensión *json* se usa para indicar que los archivos tienen formato *JSON* (de sus siglas en inglés *JavaScript Object Notation*), el cual es un formato sencillo para el intercambio de datos. El modelo es descrito de tal manera que puede ser interpretado por FEM.js para generar su representación en una escena tridimensional.

En el ejemplo 1.2.1 se presenta la solución a un ejercicio de Escamilla, 1995 usando pyFEM. En el capítulo 2 se presentan las rutinas que ejecuta pyFEM para solucionar los modelos estructurales.

Ejemplo

Resuelva completamente la cercha mostrada por el método matricial de los desplazamientos. El material es acero estructural con $E = 2040\ t/cm^2$. Las áreas están dadas entre paréntesis en cm^2.

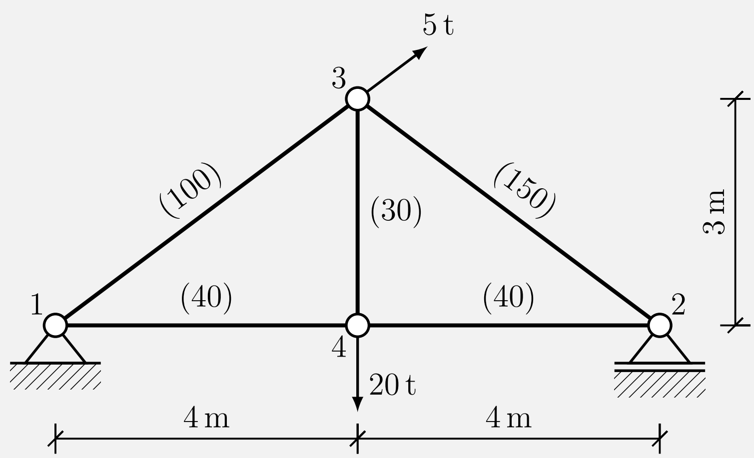

Figura 1-2: Cercha simple plana del *Ejemplo 7.1* de Escamilla, 1995.

***Solución* -** En el algoritmo 1.1 se presenta un programa de Python para analizar el modelo de la estructura usando pyFEM. Las instrucciones consisten en crear un nuevo objeto tipo `Structure`, al cual se le ha dado el nombre *model*, agregarle (a) materiales, (b) secciones transversales, (c) nodos, (d) elementos aporticados, (e) apoyos, patrones de carga y (f) cargas en los nodos, analizar el modelo y exportarlo a formato JSON.

Cuando se ejecuta la instrucción `model.solve()` pyFEM comienza a solucionar el modelo de la estructura. Los pasos que efectúa para solucionar el modelo son: (1) asignar los grados de libertad de los nodos, (2) ensamblar la matriz de rigidez del modelo de la estructura, (3) imponer las condiciones de apoyo en la matriz de rigidez del modelo, (4) ensamblar el vector de fuerzas en los nodos para cada uno de los patrones de carga, (5) imponer las condiciones de apoyo en el vector de fuerzas en los nodos para cada caso de carga, (6) encontrar los desplazamientos de los nodos para cada patrón de carga, (7) encontrar las reacciones en los apoyos para cada patrón de carga y (8) guardar la solución en los nodos y en los apoyos para cada patrón de carga.

Algoritmo 1.1: Ingreso de los datos del modelo de la estructura a *pyFEM*.

```
# create the model
model = Structure(ux=True, uy=True)

# add materials
model.add_material(key='1', modulus_elasticity=2040e4)

# add sections
model.add_section(key='1', area=030e-4)
model.add_section('2', area=040e-4)
model.add_section('3', area=100e-4)
model.add_section('4', area=150e-4)

# add joints
model.add_joint(key=1, x=0, y=0)
model.add_joint(2, 8, 0)
model.add_joint(3, 4, 3)
model.add_joint(4, 4, 0)

# add frames
model.add_frame(key='1-3', key_joint_j=1, key_joint_k=3, key_material='1'
    , key_section='3')
model.add_frame('1-4', 1, 4, '1', '2')
model.add_frame('3-2', 3, 2, '1', '4')
model.add_frame('4-2', 4, 2, '1', '2')
model.add_frame('4-3', 4, 3, '1', '1')

# add supports
model.add_support(key_joint=1, ux=True, uy=True)
model.add_support(2, ux=False, uy=True)

# add load patterns
model.add_load_pattern(key='point loads')

# add point loads
model.add_load_at_joint(key_load_pattern='point loads', key_joint=3, fx=5
     * 0.8, fy=5 * 0.6)
model.add_load_at_joint('point loads' , 4, fy=-20)

# solve the problem
model.solve()
print(model)

# export the model
model.export('example_1.json')
```

Para realizar el ensamblaje de la matriz de rigidez del modelo de la estructura y del vector de fuerzas de los nodos, pyFEM asigna números a los grados de libertad de los nodos de la estructura en el orden en que fueron ingresados; al nodo 1 se le han asignado los grados de libertad *0* y *1*, al nodo 2 los grados de libertad *2* y *3*, y así sucesivamente.

Una vez se establecen los grados de libertad de los nodos se ensambla la matriz de rigidez del modelo de la estructura. Este proceso consiste en calcular una a una las matrices de rigidez de los elementos ensamblandolas en la matriz de rigidez del modelo.

El usuario puede consultar las matrices de rigidez de cada uno de los elementos del modelo de la estructura. En (1-1) se presenta la matriz de rigidez en coordenadas locales del elemento 1-3, la cual se obtiene mediante la instrucción `model.frames['1-3'].get_local_stiffness_matrix()`.

$$\begin{bmatrix} 40800 & 0 & -40800 & 0 \\ 0 & 0 & 0 & 0 \\ -40800 & 0 & 40800 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathrm{t/m} \tag{1-1}$$

Así mismo, el usuario puede consultar las matrices de rotación de cada uno de los elementos del modelo. En (2-6) se presenta la matriz de rotación del elemento 1-3, la cual se obtiene mendiante la instrucción `model.frames['1-3'].get_rotation_matrix()`.

$$\begin{bmatrix} 0.8 & -0.6 & 0 & 0 \\ 0.6 & 0.8 & 0 & 0 \\ 0 & 0 & 0.8 & -0.6 \\ 0 & 0 & 0.6 & 0.8 \end{bmatrix} \tag{1-2}$$

El usuario tambien puede consultar las matrices de rigidez en coordenadas globales de cada uno de los elementos del modelo de la estructura. En (1-3) se presenta la matriz de rigidez en coordenadas globales del elemento 1-3, con sus respectivos grados de libertad, la cual se obtiene mediante la instrucción `model.frames['1-3'].get_global_stiffness_matrix()`.

$$\begin{array}{c} \\ \mathit{0} \\ \mathit{1} \\ \mathit{4} \\ \mathit{5} \end{array} \begin{array}{c} \begin{array}{cccc} \mathit{0} & \mathit{1} & \mathit{4} & \mathit{5} \end{array} \\ \begin{bmatrix} 26112 & 19584 & -261112 & -19584 \\ 19584 & 14688 & -19584 & -14688 \\ -26112 & -19584 & 26112 & 19584 \\ -19584 & -14688 & 19584 & 14688 \end{bmatrix} \end{array} \mathrm{t/m} \tag{1-3}$$

En (1-4), (1-5) y (1-6) se presentan las matrices de rigidez en coordenadas globales de los elementos 1-4, 3-2 y 4-3 de la estructura las cuales se obtienen con instrucciones similares a la anterior.

$$\begin{array}{c c} & \begin{array}{cccc} \mathit{0} & \mathit{1} & \mathit{6} & \mathit{7} \end{array} \\ \begin{array}{c} \mathit{0} \\ \mathit{1} \\ \mathit{6} \\ \mathit{7} \end{array} & \begin{bmatrix} 20400 & 0 & -20400 & 0 \\ 0 & 0 & 0 & 0 \\ -20400 & 0 & 20400 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \text{t/m} \tag{1-4}$$

$$\begin{array}{c c} & \begin{array}{cccc} \mathit{4} & \mathit{5} & \mathit{2} & \mathit{3} \end{array} \\ \begin{array}{c} \mathit{4} \\ \mathit{5} \\ \mathit{2} \\ \mathit{3} \end{array} & \begin{bmatrix} 39168 & -29376 & -39168 & 29376 \\ -29376 & 22032 & 29376 & -22032 \\ -39168 & 29376 & 39168 & -29376 \\ 29376 & -22032 & -29376 & 22032 \end{bmatrix} \end{array} \text{t/m} \tag{1-5}$$

$$\begin{array}{c c} & \begin{array}{cccc} \mathit{6} & \mathit{7} & 4 & \mathit{5} \end{array} \\ \begin{array}{c} \mathit{6} \\ \mathit{7} \\ \mathit{4} \\ \mathit{5} \end{array} & \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 20400 & 0 & -20400 \\ 0 & 0 & 0 & 0 \\ 0 & -20400 & 0 & 20400 \end{bmatrix} \end{array} \text{t/m} \tag{1-6}$$

El usuario también puede consultar la matriz de rigidez del modelo de la estructura mediante la instrucción `structure.get_stiffness_matrix()`. En (1-7) se presenta la matriz de rigidez de la estructura.

$$\begin{array}{c c} & \begin{array}{cccccccc} \mathit{0} & \mathit{1} & \mathit{2} & \mathit{3} & \mathit{4} & \mathit{5} & \mathit{6} & \mathit{7} \end{array} \\ \begin{array}{c} \mathit{0} \\ \mathit{1} \\ \mathit{2} \\ \mathit{3} \\ \mathit{4} \\ \mathit{5} \\ \mathit{6} \\ \mathit{7} \end{array} & \begin{bmatrix} 46512 & 19584 & 0 & 0 & -26112 & -19584 & -20400 & 0 \\ 19584 & 14688 & 0 & 0 & -19584 & -14688 & 0 & 0 \\ 0 & 0 & 59568 & -29376 & -39168 & 29376 & -20400 & 0 \\ 0 & 0 & -29376 & 22032 & 29376 & -22032 & 0 & 0 \\ -26112 & -19584 & -39168 & 29376 & 65280 & -9792 & 0 & 0 \\ -19584 & -14688 & 29376 & -22032 & -9792 & 57120 & 0 & -20400 \\ -20400 & 0 & -20400 & 0 & 0 & 0 & 40800 & 0 \\ 0 & 0 & 0 & 0 & 0 & -20400 & 0 & 20400 \end{bmatrix} \end{array} \text{t/m} \tag{1-7}$$

Una vez se ensambla la matriz de rigidez del modelo se modifica para tener en cuenta las condiciones de apoyo. Este proceso consiste en modificar las filas y las columnas asociadas a los grados de libertad restringidos. En (1-8) se presenta la matriz de rigidez sujeta a las condiciones de apoyo del modelo de la estructura la cual se obtiene mediante la instrucción `model.get_stiffness_matrix_with_support()`.

$$\begin{array}{c} \\ \mathit{0} \\ \mathit{1} \\ \mathit{2} \\ \mathit{3} \\ \mathit{4} \\ \mathit{5} \\ \mathit{6} \\ \mathit{7} \end{array}\begin{array}{c} \begin{array}{cccccccc} \mathit{0} & \mathit{1} & \mathit{2} & \mathit{3} & \mathit{4} & \mathit{5} & \mathit{6} & \mathit{7} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 59568 & 0 & -39168 & 29376 & -20400 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -39168 & 0 & 65280 & -9792 & 0 & 0 \\ 0 & 0 & 29376 & 0 & -9792 & 57120 & 0 & -20400 \\ 0 & 0 & -20400 & 0 & 0 & 0 & 40800 & 0 \\ 0 & 0 & 0 & 0 & 0 & -20400 & 0 & 20400 \end{bmatrix} \end{array} \mathrm{t/m} \tag{1-8}$$

Una vez se obtiene la matriz de rigidez modificada del modelo de la estructura se resuelve para cada uno de los patrones de carga.

Así como se deben encontrar las matrices de rigidez de cada uno de los elementos del modelo de la estructura para posteriormente ensamblarlas, se debe encontrar la acción en los nodos de cada carga. En (1-9) se presenta el vector de fuerzas nodales del modelo para el patrón de carga *point loads* mediante la instrucción `structure.load_patterns['point loads'].get_f()`.

$$\begin{array}{c} \mathit{0} \\ \mathit{1} \\ \mathit{2} \\ \mathit{3} \\ \mathit{4} \\ \mathit{5} \\ \mathit{6} \\ \mathit{7} \end{array}\begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 4 \\ 3 \\ 0 \\ -20 \end{Bmatrix} \mathrm{t} \tag{1-9}$$

Obtenido el vector de fuerzas para dicho patrón de carga se imponen las condiciones de apoyo del modelo de la estructura. Debido a que los desplazamiento en los apoyos son iguales a cero el vector de fuerzas en los nodos no varia.

Al contar con la matriz de rigidez y el vector de fuerzas en los nodos, ambos modificados por las condiciones de apoyo, se calculan los desplazamientos y las reacciones. En (1-10) y (1-11) se presentan el vector de desplazamientos y el vector de fuerzas en los nodos del modelo de la estructura para el patrón de carga *point loads*, los cuales son iguales a los presentados en Escamilla, 1995.

$$\begin{array}{c} \mathit{0} \\ \mathit{1} \\ \mathit{2} \\ \mathit{3} \\ \mathit{4} \\ \mathit{5} \\ \mathit{6} \\ \mathit{7} \end{array} \begin{Bmatrix} 0 \\ 0 \\ 1.307 \\ 0 \\ 0.645 \\ -1.337 \\ 0.654 \\ -2.317 \end{Bmatrix} 1 \times 10^{-3}\,\mathrm{m} \tag{1-10}$$

$$\begin{array}{c} \mathit{0} \\ \mathit{1} \\ \mathit{2} \\ \mathit{3} \\ \mathit{4} \\ \mathit{5} \\ \mathit{6} \\ \mathit{7} \end{array} \begin{Bmatrix} -4 \\ 7 \\ 0 \\ 10 \\ 4 \\ 3 \\ 0 \\ -20 \end{Bmatrix} \mathrm{t} \tag{1-11}$$

Cuando se ejecuta la instrucción `print(model)` pyFEM genera un informe del análisis. A continuación se presenta el informe generado para el objecto `model`.

```
Flag joint displacements
-------------------------
ux: True
uy: True
uz: False
rx: False
ry: False
rz: False

Materials
---------
label   E                       G
1              20400000.0,      0

Sections
--------
label   A               Ix      Iy      Iz
1              0.003,   0,      0,      0
```

```
2                 0.004,   0,       0,       0
3                 0.01,    0,       0,       0
4                 0.015,   0,       0,       0

Joints
------
label    x        y        z
1                 0,       0,       0
2                 8,       0,       0
3                 4,       3,       0
4                 4,       0,       0

Frames
------
label    Joint j  Joint k  material         section
1-3               1                 3                 1                           3
1-4               1                 4                 1                           2
3-2               3                 2                 1                           4
4-2               4                 2                 1                           2
4-3               4                 3                 1                           1

Supports
--------
label    ux                uy                uz                rx
    ry                rz
1                 True,    True,    False,   False,   False,   False
2                 False,   True,    False,   False,   False,   False

Load patterns
-------------
point loads:
label          fx          fy          fz          mx          my          mz
3                4.0         3.0          0           0           0           0
4                  0         -20          0           0           0           0

Displacements
-------------
point loads:
label           ux           uy           uz           rx           ry           rz
1                 +0.00000,        +0.00000,        +0.00000,        +0.00000,
            +0.00000,        +0.00000
2                 +0.00131,        +0.00000,        +0.00000,        +0.00000,
            +0.00000,        +0.00000
3                 +0.00065,        -0.00134,        +0.00000,        +0.00000,
            +0.00000,        +0.00000
4                 +0.00065,        -0.00232,        +0.00000,        +0.00000,
            +0.00000,        +0.00000
```

```
Reactions
----------
point loads:
label         fx          fy          fz          mx          my          mz
1              -4.00000,      +7.00000,      +0.00000,      +0.00000,
          +0.00000,      +0.00000
2              +0.00000,      +10.00000,     +0.00000,      +0.00000,
          +0.00000,      +0.00000
```

Cuando se ejecuta la instrucción `model.export('example_1.json')` pyFEM genera un archivo que contiene toda la información del modelo en formato JSON para ser leído por FEM.js. A continuación se presenta la información del modelo en formato JSON.


```
{
    "materials": {
        "1": {
            "E": 20400000.0,
            "G": 0
        }
    },
    "sections": {
        "1": {
            "area": 0.003,
            "Ix": 0,
            "Iy": 0,
            "Iz": 0,
            "type": "Section"
        },
        "2": {
            "area": 0.004,
            "Ix": 0,
            "Iy": 0,
            "Iz": 0,
            "type": "Section"
        },
        "3": {
            "area": 0.01,
            "Ix": 0,
            "Iy": 0,
            "Iz": 0,
            "type": "Section"
        },
        "4": {
```

```
            "area": 0.015,
            "Ix": 0,
            "Iy": 0,
            "Iz": 0,
            "type": "Section"
        }
    },
    "joints": {
        "1": {
            "x": 0,
            "y": 0,
            "z": 0
        },
        "2": {
            "x": 8,
            "y": 0,
            "z": 0
        },
        "3": {
            "x": 4,
            "y": 3,
            "z": 0
        },
        "4": {
            "x": 4,
            "y": 0,
            "z": 0
        }
    },
    "frames": {
        "1-3": {
            "j": 1,
            "k": 3,
            "material": "1",
            "section": "3"
        },
        "1-4": {
            "j": 1,
            "k": 4,
            "material": "1",
            "section": "2"
        },
        "3-2": {
            "j": 3,
            "k": 2,
            "material": "1",
            "section": "4"
```

```
        },
        "4-2": {
            "j": 4,
            "k": 2,
            "material": "1",
            "section": "2"
        },
        "4-3": {
            "j": 4,
            "k": 3,
            "material": "1",
            "section": "1"
        }
    },
    "supports": {
        "1": {
            "ux": true,
            "uy": true,
            "uz": false,
            "rx": false,
            "ry": false,
            "rz": false
        },
        "2": {
            "ux": false,
            "uy": true,
            "uz": false,
            "rx": false,
            "ry": false,
            "rz": false
        }
    },
    "load_patterns": {
        "point loads": {
            "joints": {
                "3": [
                    {
                        "fx": 4.0,
                        "fy": 3.0,
                        "fz": 3.0,
                        "mx": 0,
                        "my": 0,
                        "mz": 0
                    }
                ],
                "4": [
                    {
```

```
                    "fx": 0,
                    "fy": -20,
                    "fz": -20,
                    "mx": 0,
                    "my": 0,
                    "mz": 0
                }
            ]
        }
    }
  }
}
```

1.2.2. FEM.js

FEM.js fue desarrollado en *Three.js*. Según Dirksen, 2015, Three.js es un *API* (de sus siglas en inglés *application programming interface*) programada en *JavaScript* para *WebGL* que permite crear escenas tridimensionales en el navegador de internet. Los archivos revelantes que componen el repositorio de FEM.js son:

```
FEM.js/
├── LICENSE
├── README.md
├── css/
│   └── style.css
├── example_1.json
├── example_2.json
├── example_3.json
├── index.html
├── libs/
│   ├── CSS2DRenderer.js
│   ├── OrbitControls.js
│   ├── Projector.js
│   ├── dat.gui.min.js
│   ├── stats.js
│   └── three.js
├── main.js
└── modules/
    ├── FEM.js
    └── terminal.js
```

El archivo `LICENCE` contiene la licencia de FEM.js mientras que el archivo `README.md` contiene todas las instrucciones necesarias para ejecutar y usar FEM.js.

El archivo `index.html` define la estructura de la página web de FEM.js. En la *etiqueta* `head` se define la ubicación los archivos `three.js`, `CCS2Renderer.js`, `OrbitControls.js`, `dat.gui.min.js`, `stats.js`, el estilo de la página según el archivo `style.css` y el *módulo* `main.js`. En la etiqueta `body` se definen las secciones `renderer-output` y `console`, para mostrar la escena tridimensional y recibir las instrucciones del usuario respectivamente (véase la figura **1-3**).

Los archivos `three.js`, `CCS2Renderer.js` y `OrbitControls.js` son necesarios para renderizar gráficos con WebGL, asociar objetos de la escena con etiquetas html y manipular la cámara. Estos archivos hacen parte del repositorio del proyecto Three.js alojado en GitHub (https://github.com/mrdoob/three.js/).

Los archivos `dat.gui.min.js` y `stats.js` permiten crear interfaces gráficas de usuario que cambian el valor de las variables y un monitor del desemepeño del código respectivamente. Estos archivos hacen parte de los repositorios *dat.gui* y *stats.js* alojados en GitHub (https://github.com/dataarts/dat.gui y https://github.com/mrdoob/stats.js/).

El archivo `style.css` define la presentación de las etiquetas html de la página web.

El archivo `main.js` define las funciones del archivo `FEM.js` que ejecuta `terminal.js` y algunos *eventos* para que todos los elementos de la página funcionen adecuadamente. El archivo `terminal.js` define una serie de funciones para interpretar y ejecutar las instrucciones que ingrese el usuario.

El archivo `FEM.js` contiene la configuración por defecto del programa, la descripción del panel lateral derecho y todas las funciones que hacen posible que el usuario pueda interactuar con el modelo.

Los archivos `example_1.json`, `example_2.json` y `example_3.json` almacenan los modelos de tres de los ejemplos presentados en Escamilla, 1995 que han sido generados con pyFEM.

FEM.js puede representar cualquiera de estos archivos al ejecutar la función `open()` con el nombre del archivo entre comillas dobles o sencillas como dato de entrada. En la figura **1-3** se presenta FEM.js con el archivo `example_2.json` abierto ejecutándose en el navegador de internet Firefox.

Así mismo, es capaz de ejecutar todas las funciones que se definan con `add_funcion()` del archivo `terminal.js`. Esta función recibe como parámetros el nombre de la función y un

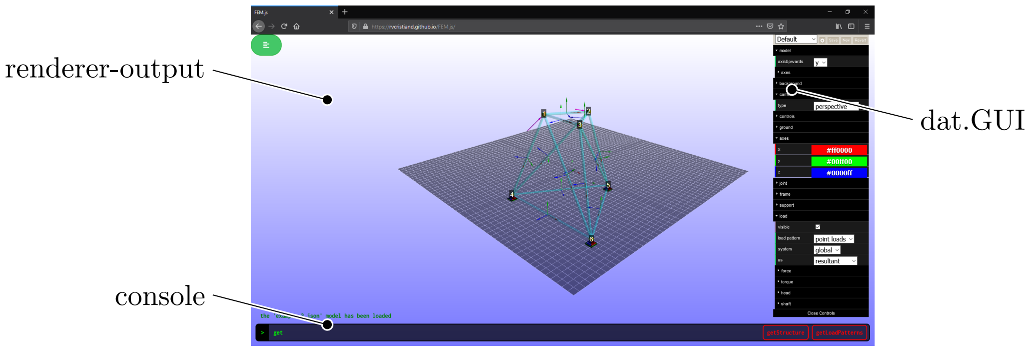

Figura 1-3: FEM.js ejecutándose en Firefox

objecto el cual debe definir la propiedad `func`. El nombre de la función se usa para llamar al parámetro `func` con los valores seprados por comas ingresados por el usuario.

A continuación se presentan una lista de las funciones definidas en `main.js`.

- addFrame,
- removeFrame,
- addSection,
- addRectangularSection,
- removeSection,
- addMaterial,
- removeMaterial,
- addJoint,
- removeJoint,
- setFrameView,
- showJointsLabel,
- hideJointsLabel,
- showFramesLabel,
- hideFramesLabel,
- setUpwardsAxis,
- setView,
- open,
- getStructure,
- getLoadPatterns.

Aunque el nombre de la función no tenga que ser necesariamente igual al del parámetro `func`, todos los nombres de la lista coínciden con funciones definidas en el archivo `FEM.js` (aun cuando no es necesario que estén definidas ahí).

La descripción de los parámetros de entrada de cada una de estás funciones se encuentran en el archivo `README.md`. A partir de dichas instrucciones es posible generar el modelo tridimensional de la estructura. Por ejemplo, para generar el modelo de la estructura del ejemplo 1.2.1 se deben ingresar las siguientes instrucciones

```
addMaterial(1, 2040e4)

addSection(1)
addSection(2)
addSection(3)
addSection(4)

addJoint(1, 0, 0, 0)
addJoint(2, 8, 0, 0)
addJoint(3, 4, 3, 0)
addJoint(4, 4, 0, 0)

addFrame(1-3, 1, 3, 1, 3)
addFrame(1-4, 1, 4, 1, 2)
addFrame(3-2, 3, 2, 1, 4)
addFrame(4-2, 4, 2, 1, 2)
addFrame(4-3, 4, 3, 1, 1)

addSupport(1, true, true)
addSupport(2, false, true)

addLoadPattern('point loads')

addLoadAtJoint('point loads', 3, 4, 3)
addLoadAtJoint('point loads', 4, 0, -20)
```

En la figura **1-4** se presenta FEM.js después de ejecutar los anteriores comandos.

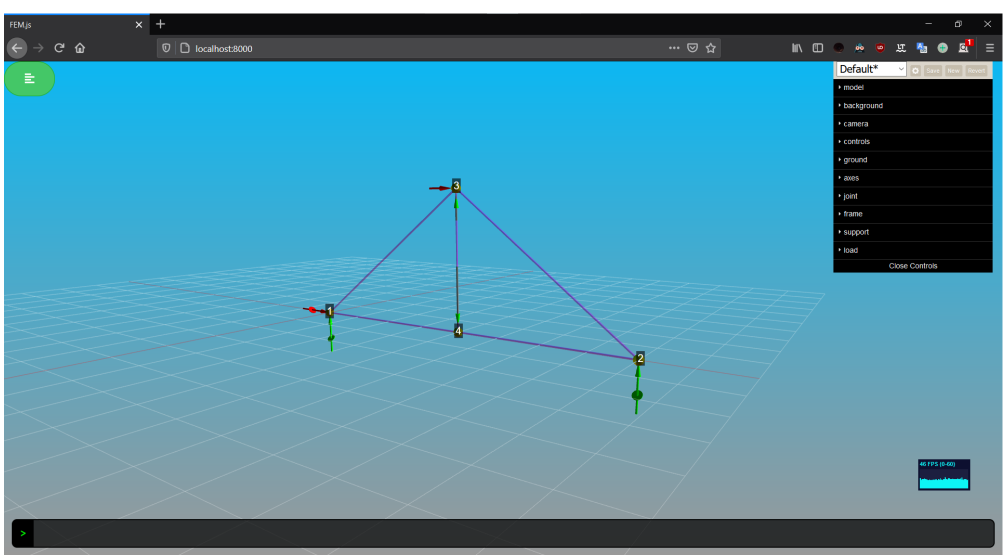

Figura 1-4: Ejemplo 1.2.1 modelado en FEM.js.

dat.gui

El panel lateral derecho de FEM.js fue desarrollado con *dat.GUI* para que el usuario pueda personalizar la escena. Este panel está agrupado en las siguientes categorias:

- model,
- background,
- camera,
- controls,
- ground,
- axes,
- joint,
- frame,
- support,
- load.

En la sección `model` se establece la orientación del modelo al definir uno de los ejes principales del modelo que apunta hacía la parte superior de la pantalla y una subsección llamada *axes*. En esta sección se define el tamaño y visibilidad de los ejes principales del modelo y dos subsecciones llamadas *head* y *shaft*. En estas susecciones se define la geometría de la cabeza y la cola de los vectores de los ejes principales del modelo.

En la sección `background` se establen dos colores para generar el fondo de la escena en gradiente. El color *top* define el color para la parte superior del fondo de la escena mientras que el color *bottom* define el color para la parte inferior.

En la sección `camera` se establece el tipo de proyección de la cámara pudiéndose elegir entre perspectiva y ortogonal. En la figura **1-5** se presenta el modelo del archivo `example_2.json` en proyección ortogonal.

En la sección `controls` se establece el comportamiento de los controles de FEM.js. Ahí se define la velocidad con la que estos hacen rotar, hacen *zoom*, desplazan la escena, si se desplaza la escena paralelo al plano del modelo o al plano de la proyección y una subsección llamada `damping`. En esta subsección se define si se adiciona un *amortiguamiento* a la rotación y la intensidad del mismo.

En la sección `ground` se define la visibilidad y el tamaño del conjunto de elementos *plano* y *grilla* así como dos secciones llamadas `plane` y `grid`. En la sección `plane` se define la visibilidad, el color, la transparencia y la opacidad del plano del modelo mientras que en la sección `grid` se define la visibilidad, el número de divisiones y los colores de las divisiones mayores y menores de la grilla.

En la sección `axes` se definen tres colores los cuales se asocian a los ejes x, y y z. Estos colores establecen los colores de los ejes globales y locales, los apoyos y las cargas. En la

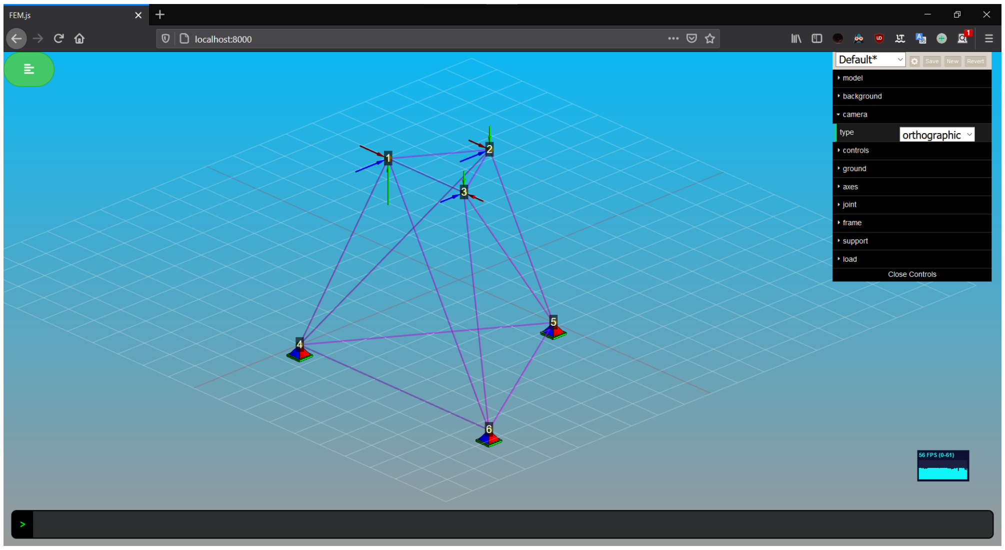

Figura 1-5: FEM.js en proyección ortogonal.

figura **1-6** se presenta el modelo del archivo `example_3.json` con una definición alternativa de dichos colores.

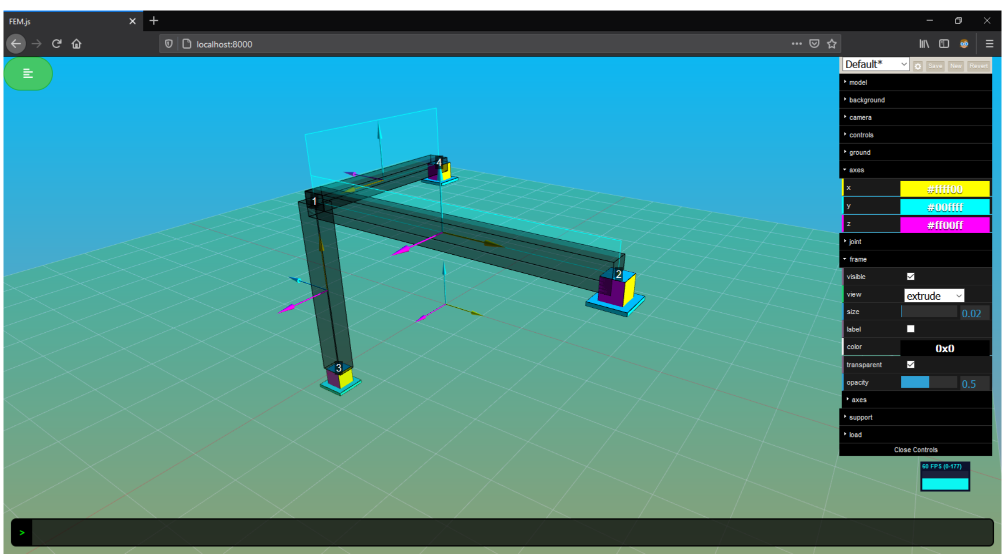

Figura 1-6: Colores alternativos para los elementos asociados a los ejes x, y y z.

En la sección `joint` se define la visibilidad, el tamaño, el color, la transparencia y la opacidad de los nodos del modelo. Así mismo se define la visibilidad de los *labels* de los nodos.

En la sección `frame` se define la visibilidad, la vista (*extruida* o en *palillo*), el tamaño, el color, la transparencia y la opacidad de los elementos aporticados del modelo. Así mismo se define la visibilidad de los *labels* de estos elementos y una sección llamada `axes`, similar a la que se encuentra en la sección `model`, con la diferencia que esta establece la visibilidad y el tamaño de los ejes locales de los elementos aporticados. En la figura **1-7** se presenta el modelo del archivo `example_3.json` en *estructura de palillo*.

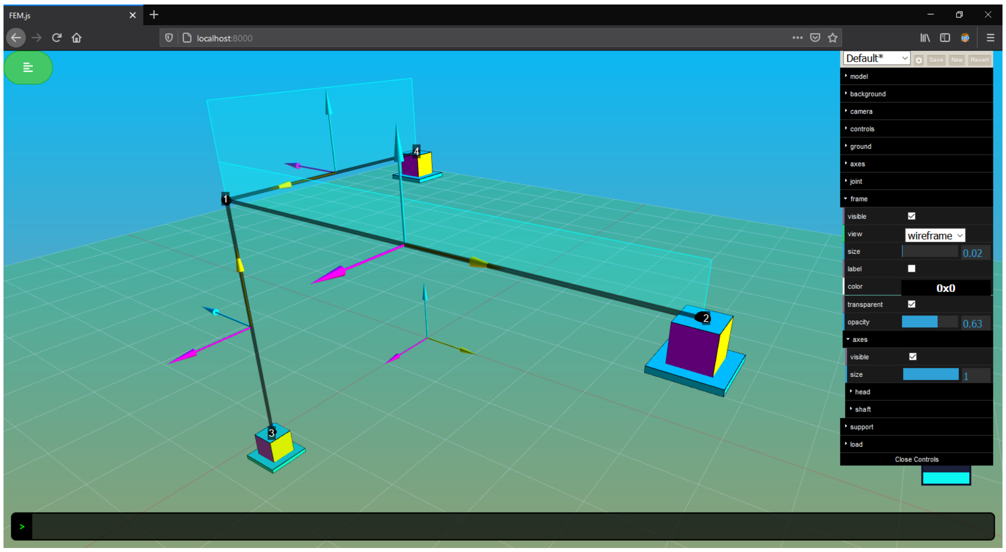

Figura 1-7: Vista del modelo como *estructura de palillo*.

En la sección `support` se define la visibilidad, el *modo* de los apoyos del modelo y dos secciones llamadas `analytical` y `space`. Los modos de los apoyos pueden ser *space* o *analytical*, en donde estos se representan con alguna de las analogías usadas en la literatura para representar apoyos o mediante vectores con un disco inclinado en la mitad de las colas.

En la sección `analytical` se definen tres secciones llamadas `head`, `shaft` y `restraint`, con las cuales se puede definir la geometría de los vectores con colas rectas o curvas (para representar restricciones a la traslación o a la rotación respectivamente) que tienen un un disco inclinado en la mitad de la cola.

En la sección `space` se definen tres secciones llamadas `foundation`, `pedestal` y `pin`, con las cuales se puede definir la geometría de los elementos *fundación*, *pedestal* o *rótula*, usados para representar los apoyos como elementos espaciales. Cuando se restringen todas las traslaciones y rotaciones el apoyo se resepresenta mediante un pedestal y una fundación, mientras que si se restrigen solo las translaciones el apoyo se representa por un pedestal y una pirámide con base cuadrada. Estos apoyos toman los colores definidos en la sección `axes` de manera conveniente.

En la figura **1-8** se presenta el modelo del archivo `example_3.json` con los apoyos en modo *analytical*.

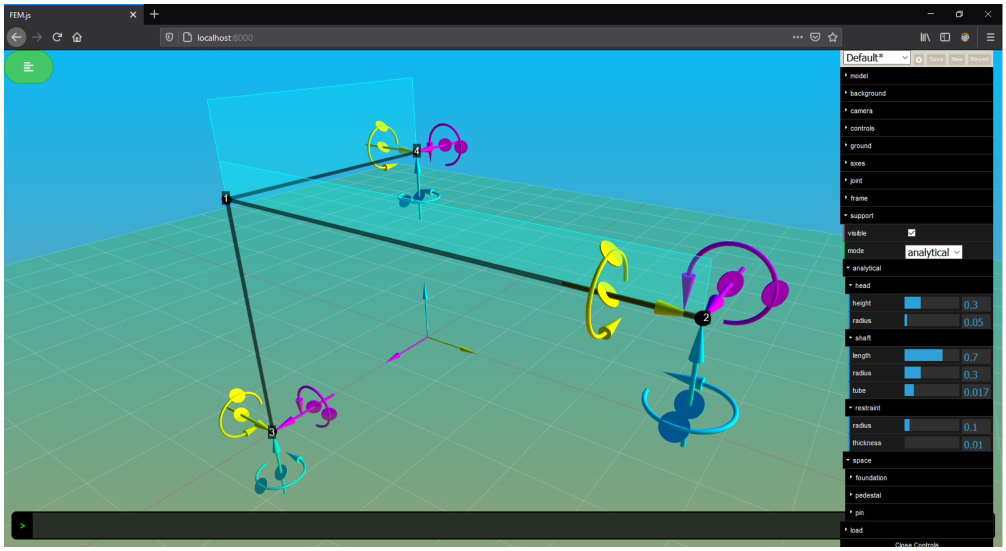

Figura 1-8: Apoyos del modelo en modo *analytical*.

En la sección `load` se define la visibilidad, el patrón de cargas, el sistema de coordenadas de referencia y como se representan las cargas del modelo, así como cuatro secciones con los nombres `force`, `torque`, `head` y `shaft`. En el momento únicamente se cuenta con el sistemda de refencia global para representar las cargas mientras que se pueden represetar como resultantes o componentes, aunque esta última opción actualmente sólo esta disponible para las cargas puntuales.

En las secciones `force`, `torque`, `head` y `shaft` se definen las dimensiones y el color de los elementos que representan las cargas. El tamaño de los diferentes elementos para representar las cargas se escalan en función de valor que estas representen.

2 pyFEM

pyFEM es un programa de computador desarrollado en Python para analizar linealmente estructuras aporticadas tridimensionales sometidas a cargas estáticas. Una copia del programa se encuentra alojada en la página web de GitHub https://github.com/rvcristiand/pyFEM.

Los principales archivos del programa son:

```
pyFEM/
├── LICENSE
├── README.md
├── example_1.json
├── example_2.json
├── example_3.json
├── pyFEM/
│   ├── classtools.py
│   ├── core.py
│   └── primitives.py
└── test/
    ├── space_frame.py
    └── trusses.py
```

El usuario puede generar objetos de la clase `Structure`, definida en el archivo `core.py`, para describir y analizar linealmente modelos de estructuras aporticadas tridimensionales sometidas a fuerzas estáticas. En la figura **2-1** se presentan los métodos y atributos de esta clase.

El constructor de la clase `Structure` recibe seis argumentos de entrada opcionales, uno para cada grado de libertad (tres translaciones y tres rotaciones), los cuales tienen `False` como valor por defecto. Cuando el usuario crea un objeto de esta clase debe indicar qué grados de libertad quiere tener en cuenta para analizar el modelo.

En el algoritmo 2.1 se presenta el constructor de la clase `Structure`. El constructor de la clase le asigna a los atributos `ux`, `uy`, `uz`, `rx`, `ry` y `rz` los respectivos valores de los argumentos de entrada. A los demás atribututos les asigna un diccionario vacío.

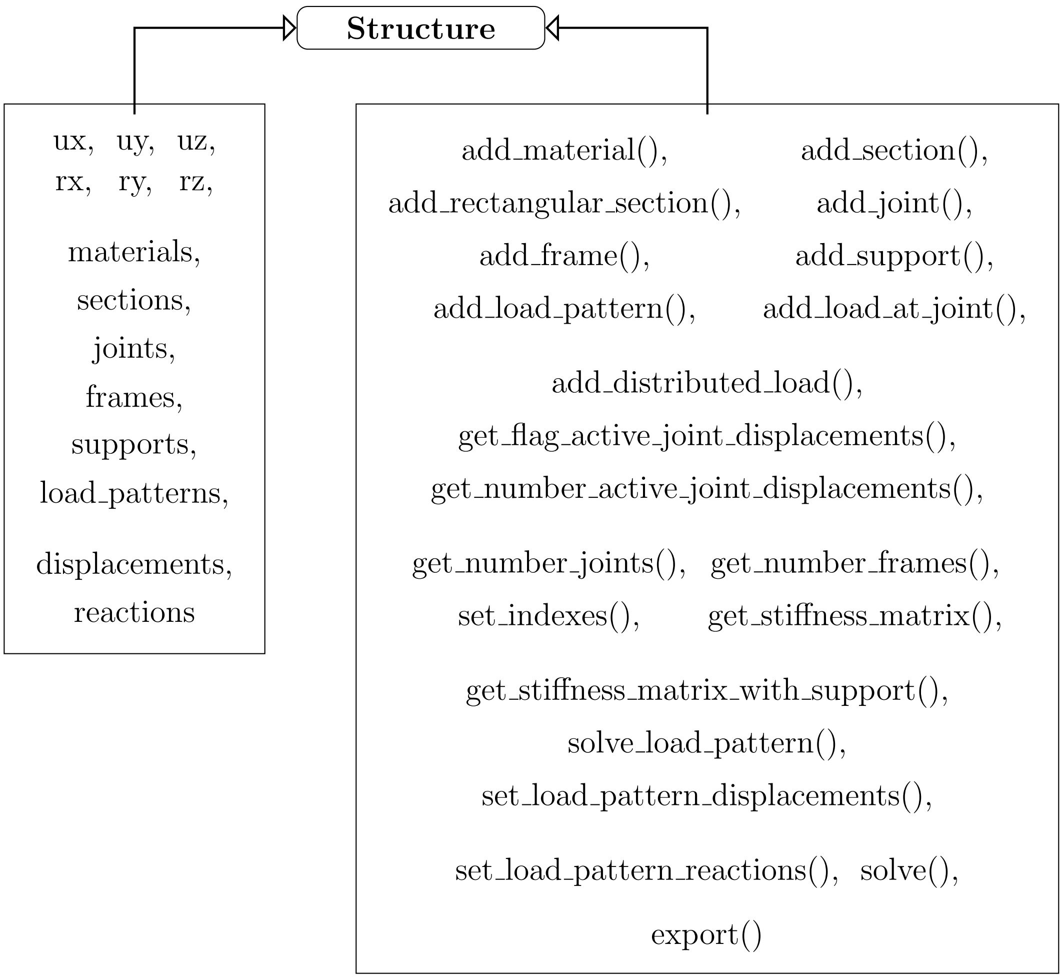

Figura 2-1: Métodos y atributos de la clase `Structure`.

Algoritmo 2.1: Constructor de la clase `Structure`.

```
def __init__(self,ux=False,uy=False,uz=False,rx=False,ry=False,rz=False):
    """
    Instantiate a Structure object

    Parameters
    ----------
    ux : bool
        Flag translaction along 'x' axis activate.
    uy : bool
        Flag translaction along 'y' axis activate.
    uz : bool
        Flag translaction along 'z' axis activate.
    rx : bool
        Flag rotation around 'x' axis activate.
    ry : bool
        Flag rotation around 'y' axis activate.
```

```
    rz : bool
        Flag rotation around 'z' axis activate.
    """
    # flag active joint displacements
    self.ux = ux
    self.uy = uy
    self.uz = uz
    self.rx = rx
    self.ry = ry
    self.rz = rz

    # dict materials and sections
    self.materials = {}
    self.sections = {}

    # dict joints and frames
    self.joints = {}
    self.frames = {}

    # dict supports
    self.supports = {}

    # dict load patterns
    self.load_patterns = {}

    # dict displacements
    self.displacements = {}

    # dict reactions
    self.reactions = {}
```

El usuario puede describir el modelo con los objetos tipo `Structure` agregándole objetos que representan materiales, secciones transversales, nodos, elementos aporticados, apoyos, patrones de carga, fuerzas aplicadas en los nodos y cargas distribuidas en los elementos aporticados, mediante los métodos que comienzan con *add*.

Estos métodos reciben uno o varios argumentos de entrada obligatorios y una serie de argumentos de entrada opcionales para crear los objectos y almacenarlos en los respectivos diccionarios del objeto tipo `Structure`.

Dichos métodos son similares entre sí, solo cambia el diccionario al cual se le está agregando la nueva entrada y el objecto que se está creando. Por ejemplo, en el algoritmo 2.2 se presenta el método `add_material()`. Los argumentos de entrada `*args` y `**kwargs` son pasados al constructor de la clase `Material` para crear un objeto tipo `Material`, mientras que el argumento `key` es usado como llave para almacenar dicho objeto en el diccionario `materials`.

Algoritmo 2.2: Método `add_material()` de la clase `Structure`.

```
def add_material(self, key, *args, **kwargs):
    """
    Add a material

    Parameters
    ----------
    key : immutable
        Material's key.
    """
    self.materials[key] = Material(*args, **kwargs)
```

En el caso de los métodos `add_section()`, `add_rectangular_section()`, `add_joint()` y `add_load_pattern()`, el diccionario ya no es `materials` sino `sections`, `joints` o `load_patterns`, según corresponda, y el objeto a crear ya no es de tipo `Material` sino de tipo `Section`, `RectangularSection`, `Joint` o `LoadPattern`, respectivamente.

El método `add_frame()` permite agregar objetos tipo `Frame` de manera similar a como lo hace el método `add_material()`, con la diferencia que este método recibe como argumentos de entrada las llaves con las que fueron creados el nodo cercano, el nodo lejano, el material y la sección transversal.

En el algoritmo 2.3 se presenta el método `add_frame()`. Las llaves del material, de la sección transversal y de los nodos se utilizan para recuperan los objetos relacionados en los diferentes diccionarios del objeto tipo `Structure` para crea el objeto tipo `Frame`.

Algoritmo 2.3: Método `add_frame()` de la clase `Structure`.

```
def add_frame(self, key, key_joint_j, key_joint_k, key_material, key_section):
    """
    Add a frame

    Parameters
    ----------
    key : immutable
        Frame's key.
    key_joint_j : immutable
        Joint j's key.
    key_joint_k : immutable
        Joint k's key.
    key_material : immutable
        Material's key.
    key_section : immutable
```

```
            Section's key.
        """
        self.frames[key] = Frame(self.joints[key_joint_j], self.joints[key_joint_k
    ], self.materials[key_material], self.sections[key_section])
```

El método `add_support()` es similar a los anteriores, con la diferencia que los objeto tipo `Joint` son usados como llaves para almacenar los objeto tipo `Support`, tal como se presenta en el algoritmo 2.4.

Algoritmo 2.4: Método `add_support()` de la clase `Structure`.

```
def add_support(self, key_joint, *args, **kwargs):
    """
    Add a support

    Parameters
    ----------
    key_joint : immutable
        Joint's key.
    """
    self.supports[self.joints[key_joint]] = Support(*args, **kwargs)
```

Por su parte, los métodos `add_load_at_joint()` y `add_distributed_load()` reciben dos argumentos de entrada obligatorios y una serie de argumentos de entrada opcionales. El primer argumento de entrada obligatorio es la llave con la que se creó el objeto tipo `LoadPattern` y el segundo es la llave con el que se creó el objeto tipo `Joint` o el objeto tipo `Frame`, respectivamente.

En el algoritmo 2.5 se presenta el método `add_load_at_joint()`. Con la llave del patrón de carga se recupera el objeto tipo `LoadPattern` mientras que con la llave del nodo se recupera el objeto tipo `Joint`. El objeto tipo `Joint` y los demás argumentos de entrada opcionales son pasados al método `add_point_load_at_joint()`.

Algoritmo 2.5: Método `add_load_at_joint()` de la clase `Structure`.

```
def add_load_at_joint(self, key_load_pattern, key_joint, *args, **kwargs):
    """
    Add a point load at joint

    Parameters
    ----------
    key_load_pattern : immutable
        Load pattern's key.
    key_joint : immutable
```

```
        Joint's key,
    """
    self.load_patterns[key_load_pattern].add_point_load_at_joint(self.joints[
    key_joint], *args, **kwargs)
```

En el caso del método `add_distributed_loads()`, el objeto tipo `Frame` y los demás argumentos de entrada opcionales son pasados al método `add_distributed_load()` del objeto tipo `LoadPattern`.

Cuando el usuario termina de describir el modelo puede ejecutar el método `solve()` de la clase `Structure` para analizarlo. pyFEM soluciona la estructura sometida a los diferentes patrones de carga, almacenando los resultados de los vectores de desplazamientos y fuerzas en los nodos en los atributos `displacements` y `reactions`, respectivamente.

En las siguientes secciones se presenta la implementación de las clases con las que el usuario puede describir el modelo (`Material`, `Section`, `Joint`, etc.) y después la implementación de los demás métodos de la clase `Structure`.

2.1. Clases

Además de la clase `Structure`, definida en el archivo `core.py`, pyFEM define otras clases en los archivos `primitives.py` y `classtools.py`.

En el archivo `primitives.py` se definen todas las clases que permiten describir el modelo, es decir:

- `Material`,
- `Section`,
- `RectangularSection`,
- `Joint`,
- `Frame`,
- `Support`,
- `LoadPattern`,
- `PointLoad`,
- `DistributedLoad`,
- `Displacement`,
- `Reaction`.

En el archivo `classtools.py` se define la clase `AttrDisplay` y la *metaclase* `UniqueInstance`. La clase `AttrDisplay` implementa una representación de los objetos más cómoda para los usuarios, mientras que la metaclase `UniqueInstance` no permite crear objetos con los mismos atributos de otros objetos de la misma clase.

A continuación se presenta la implementación de todas las clases de pyFEM.

2.1.1. Material

La clase `Material` representa un material líneal elástico al definir los valores del módulo de Young y del módulo a cortante.

En el algoritmo 2.6 se presenta la implementación de la clase `Material`. Se asigna la tupla `('E', 'G')` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia. Esto como mecánismo de optimización.

Según Lutz, 2013, asignar un diccionario en el espacio de nombres para cada objeto instanciado puede ser costoso, en términos de memoria, si muchas instancias son creadas y solo se requiere un par de atributos. Para ahorrar memoria, en lugar de asignar un diccionario por instancia, Python reserva el espacio suficiente en cada instancia para guardar un valor por cada atributo del *slot*.

El constructor de la clase recibe los argumentos de entrada `modulus_elasticity` y `shearing_modulus_elasticity`, los cuales tienen 0 como valor por defecto. Estos valores son pasados a los atributos `E` y `G` del objeto tipo `Material` respectivamente.

Algoritmo 2.6: Clase `Material` implementada en el archivo `primitives.py`.

```
class Material(AttrDisplay):
    """
    Linear elasic material

    Attributes
    ----------
    E : float
        Young's modulus.
    G : float
        Shear modulus.
    """
    __slots__ = ('E', 'G')

    def __init__(self, modulus_elasticity=0, shearing_modulus_elasticity=0):
        """
        Instantiate a Material object

        Parameters
        ----------
```

```
        modulus_elasticity : float
            Young's modulus.
        shearing_modulus_elasticity : float
            Shear modulus.
        """
        self.E = modulus_elasticity
        self.G = shearing_modulus_elasticity
```

2.1.2. Section

La clase `Section` representa la sección transversal de los elementos aporticados de manera general, al definir los valores del área transversal, de la constante de torsión y de las inercias principales alrededor de los ejes principales.

En el algoritmo 2.7 se presenta la implementación de la clase `Section`. Así como se asignó una tupla al atributo `__slots__` de la clase `Material`, como mecánismo de optimización, se asigna una tupla al atributo `__slots__` de la clase `Section` con los elementos `'A'`, `'Ix'`, `'Iy'` y `'Iz'`.

El constructor de la clase recibe los argumentos de entrada `area`, `torsion_constant`, `moment_inertia_y` y `moment_inertia_z`, los cuales tienen 0 como valor por defecto. Estos valores son pasados a los atributos `A`, `Ix`, `Iy` y `Iz`, respectivamente.

Algoritmo 2.7: Clase `Section` implementada en el archivo `primitives.py`.

```
class Section(AttrDisplay):
    """
    Cross-sectional area

    Attributes
    ----------
    A : float
        Cross-sectional area.
    Ix : float
        Inertia around axis x-x.
    Iy : float
        Inertia around axis y-y.
    Iz : float
        Inertia around axis z-z.
    """
    __slots__ = ('A', 'Iy', 'Iz', 'Ix')

    def __init__(self, area=0, torsion_constant=0, moment_inertia_y=0,
    moment_inertia_z=0):
```

```
        """
        Instantiate a Section object

        Parameters
        ----------
        area : float
            Cross-sectional area.
        torsion_constant : float
            Inertia around axis x-x.
        moment_inertia_y : float
            Inertia around axis y-y.
        moment_inertia_z : float
            Inertia around axis z-z.
        """
        self.A = area
        self.Ix = torsion_constant
        self.Iy = moment_inertia_y
        self.Iz = moment_inertia_z
```

2.1.3. RectangularSection

La clase `RectangularSection` representa la sección transversal de forma rectángular de los elementos aporticados, al definir los valores de la base y del alto de la figura.

En el algoritmo 2.8 se presenta la implementación de la clase `RectangularSection`. Esta clase hereda todos los métodos y atributos de la clase `Section`, para evitar duplicar el código a lo largo del programa, al pasar dicha clase como argumento de entrada cuando se construye la clase `RectangularSection`.

Al atributo `__slots__` de la clase `RectangularSection` se le asigna una tupla con las entradas `'width'` y `'height'`. Python no solo limita las instancias de esta clase a los atributos `width` y `height`, sino que se extiende a los elementos definidos en el atributo `__slots__` de la clase `Section`.

Según Lutz, 2013, como las variables `__slots__` son atributos a nivel de clases, los objetos instanciados adquieren la unión de todas las entradas en todos los atributos `__slots__` de la clase y sus *super clases*.

El constructor de la clase recibe los argumentos de entrada `width` y `height`, los cuales no tiene ningún valor por defecto (a diferencia de los argumentos de entrada del constructor de la clase `Section`). Los valores de los argumentos de entrada son asignados a los respectivos atributos de los objeto tipo `RectangularSection`.

Se asume que el valor del parámetro `width` corresponde a la dimensión del elemento aporticado de sección transversal rectangular a lo largo del eje y del sistema de coordenadas local, mientras que el parámetro `heigth` corresponde a la dimensión del elemento aporticado a lo largo del eje z.

Teniendo en cuenta esto se cálcula el área, la constante de torsión y los momentos de inercia alrededor de los ejes y y z. Para calcular la constante de torsión se utiliza la expresión (2-1), la misma que se presenta en Escamilla, 1995,

$$I_{xx} = \left(\frac{1}{3} - 0.21 \frac{a}{b} \left(1 - (1/12)(a/b)^4 \right) \right) ba^3 \tag{2-1}$$

donde `a` es la dimensión menor del rectángulo y `b` su dimensión mayor.

Finalmente, las propiedades de la sección transversal son pasadas al constructor de la clase `Section` para asignárselas a los atributos del objeto tipo `RectangularSection`. Esto se hace mediante la función `super()` que trae Python por defecto, la cual genera una referencia a la *clase padre*, en este caso, la clase `Section`.

Algoritmo 2.8: Clase `RectangularSection` implementada en el archivo `primitives.py`.

```
class RectangularSection(Section):
    """
    Rectangular cross-section

    Attributes
    ----------
    width : float
        Width rectangular cross section.
    height : float
        Height rectangular cross section.
    A : float
        Cross-sectional area.
    Ix : float
        Inertia around axis x-x.
    Iy : float
        Inertia around axis y-y.
    Iz : float
        Inertia around axis z-z.
    """
    __slots__ = ('width', 'height')

    def __init__(self, width, height):
        """
        Instantiate a rectangular section object
```

```
        Parameters
        ----------
        width : float
            Width rectangular cross section.
        height : float
            Height rectangular cross section.
        """
        self.width = width
        self.height = height

        a = min(width, height)
        b = max(width, height)
        area = width * height
        torsion_constant = (1/3 - 0.21 * (a / b) * (1 - (1/12) * (a/b)**4)) *
    b * a ** 3
        moment_inertia_y = (1 / 12) * width * height ** 3
        moment_inertia_z = (1 / 12) * height * width ** 3

        super().__init__(area, torsion_constant, moment_inertia_y,
    moment_inertia_z)
```

2.1.4. Joint

La clase `Joint` representa nodos de la estructura, al definir sus coordenadas en el sistema de coordenadas global.

En el algoritmo 2.9 se presenta la implementación de la clase `Joint`. Como mecánismo de optimización, se asigna una tupla con los elementos `x`, `y` y `z` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia.

El constructor de la clase recibe tres argumentos de entrada opcionales, uno para cada una de las coordenadas, los cuales tienen 0 como valor por defecto. Estos valores son pasados a los atributos `x`, `y` y `z` del objeto respectivamente.

Finalmente, la clase `Joint` implementa el método `get_coordinate()` que generar una *array* con las coordenadas del objeto.

Algoritmo 2.9: Clase `Joint` implementada en el archivo `primitives.py`.

```
class Joint(AttrDisplay, metaclass=UniqueInstances):
    """
    End of frames
```

```
    Attributes
    ----------
    x : float
        X coordinate.
    y : float
        Y coordinate.
    z : float
        Z coordinate.

    Methods
    -------
    get_coordinate()
        Return joint's coordinates.
    """
    __slots__ = ('x', 'y', 'z')

    def __init__(self, x=0, y=0, z=0):
        """
        Instantiate a Joint object

        Parameters
        ----------
        x : float
            X coordinate.
        y : float
            Y coordinate.
        z : float
            Z coordinate.
        """
        self.x = x
        self.y = y
        self.z = z

    def get_coordinate(self):
        """Get coordinates"""
        return np.array([self.x, self.y, self.z])
```

2.1.5. Frame

La clase `Frame` representa los elementos aporticados de la estructura, al definir sus nodos, material y sección tranversal. En la figura **2-2** se presentan los métodos y atributos de esta clase.

En la figura **2-3** se presenta un elemento aporticado i con sus nodos j y k empotrados. El sistema de coordenadas local del elemento tiene como origen el nodo j; el eje x coincide con el eje centroidal del elemento y es positivo en el sentido del nodo j al nodo k.

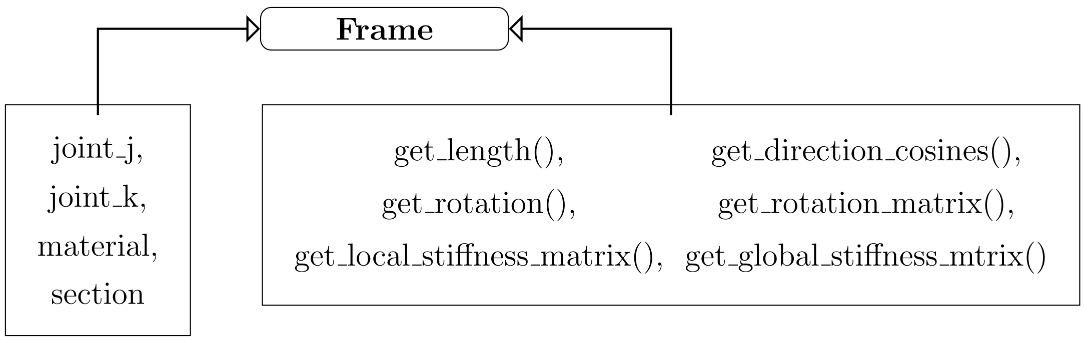

Figura 2-2: Métodos y atributos de la clase `Frame`.

Los ejes y y z son los ejes principales del elemento de manera que los planos xy y zx son los planos principales de flexión. Se asume que el centro de cortante y el centroide del elemento coinciden de tal forma que la flexión y la torsión se presentan una independiente de la otra.

Los grados de libertad se numeran del 1 al 12, empezando por las translaciones y las rotaciones del nodo j, tomados en orden x, y, z respectivamente.

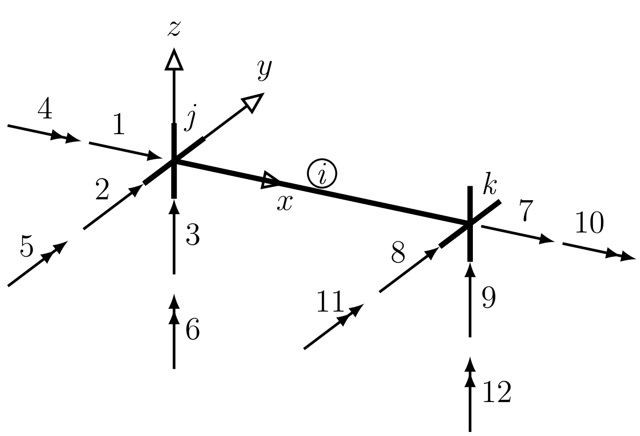

Figura 2-3: Elemento aporticado y su sistema de coordenadas local.

Como mecánismo de optimización, se asigna una tupla con los elementos `joint_j`, `joint_k`, `material` y `section` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia.

En el algoritmo 2.10 se presenta el constructor de la clase `Frame`. El constructor de la clase `Frame` recibe cuatro argumentos de entrada; para el nodo cercano, el nodo lejano, el material y la sección, los cuales tiene `None` como valor por defecto. Los argumentos de entrada son pasados a los atributos `joint_j`, `joint_k`, `material` y `section` respectivamente.

Algoritmo 2.10: Constructor de la clase `Frame`.

```
def __init__(self, joint_j=None, joint_k=None, material=None, section=None):
    """
    Instantiate a Frame object

    Parameters
    ----------
    joint_j : Joint
        Near Joint object.
    joint_k : Joint
        Far Joint object.
    material : Material
        Material object.
    section : Section
        Section object.
    """
    self.joint_j = joint_j
    self.joint_k = joint_k
    self.material = material
    self.section = section
```

A continuación se presentan los métodos de la clase `Frame`, con los cuales se puede, entre otras cosas, calcular la matriz de rigidez de los elementos tipo `Frame`.

get_length()

El método `get_length()` de la clase `Frame` permite calcular la longitud de los elementos aporticados representado por objetos tipo `Frame`.

En el algoritmo 2.11 se presenta la implementación del método `get_length()`. El método calcula la distancia que hay entre las coordenadas de los nodos del elemento aporticado. Para esto llama la función `distance.euclidean()` con las coordenadas de los objeto tipo `Joint`, las cuales obtiene con el método `get_coordinate()` (véase el algoritmo 2.9).

Según Virtanen et al, 2020, esta función calcula la distancia euclidiana entre dos *arrays* u y v de una dimensión como

$$||u-v||_2 = \left(\sum w_i |(u_i - v_i)|^2\right)^{1/2} \tag{2-2}$$

donde `w` es un *array* que toma para cada entrada un peso de 1 por defecto.

Algoritmo 2.11: Método `get_length()` de la clase `Frame`.

```
def get_length(self):
    """Get length"""
    return distance.euclidean(self.joint_j.get_coordinate(), self.joint_k.
   get_coordinate())
```

get_direction_cosines()

El método `get_direction_cosines()` de la clase `Frame` permite calcular los cosenos directores del eje x del sistema de coordenadas local de los elementos aporticados, representados por objetos tipo `Frame`, en el sistema de coordenadas global.

En el algoritmo 2.12 se presenta la implementación del método `get_direction_cosines()`. El método resta las coordenadas de los nodos del elemento aporticado y almacena el resultado en la variable `vector`. Después divide cada uno de los elementos de `vector` por la norma de dicha variable, calculada mediante la función `linalg.norm()`.

Según Harris et al, 2020, esta función calcula la norma de un vector como

$$||A||_F = \left[\sum_{i,j} abs(a_{i,j})^2\right]^{1/2} \tag{2-3}$$

donde $a_{i,j}$ es el elemento del vector en la posición i, j.

Algoritmo 2.12: Método `get_direction_cosines()` de la clase `Frame`.

```
def get_direction_cosines(self):
    """Get direction cosines"""
    vector = self.joint_k.get_coordinate() - self.joint_j.get_coordinate()

    return vector / linalg.norm(vector)
```

get_rotation()

El método `get_rotation()` de la clase `Frame` permite calcular la rotación de los elementos aporticados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas global.

Según el *teorema de rotación de Euler* (véase Akademiia nauk SSSR., 1763), siempre es posible encontrar un diámetro de una esfera cuya posición es la misma después de rotar la esfera alrededor de su centro, por lo que cualquier secuencia de rotaciones de un sistema coordenado tridimensional es equivalente a una única rotación alrededor de un eje que pase por el origen.

El ángulo θ y el vector n que definen la rotación del eje x del sistema de coordenadas global hacia el eje x_m del sistema de coordenadas local de un elemento aporticado se puede calcular como

$$\begin{aligned} \mathbf{n} &= (1,0,0) \times \mathbf{x_m} \\ \theta &= \arccos((1,0,0) \cdot \mathbf{x_m}) \end{aligned} \tag{2-4}$$

Según Dunn, 2002, la rotación de un sistema de coordenadas tridimensionales alrededor del eje $\mathbf{n}$ una cantidad θ se puede describir mediante un *cuaternión* como

$$\mathbf{q} = \begin{bmatrix} cos(\theta/2) & sin(\theta/2)\mathbf{n} \end{bmatrix} \tag{2-5}$$

y se puede obtener la matriz de rotación a partir de un cuaternión de la siguiente manera

$$\mathbf{R} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix} \tag{2-6}$$

donde w es la parte escalar y x, y y z la parte vectorial del cuaternión.

En el algoritmo 2.13 se presenta la implementación del método `get_rotation()`. El método calcula el *cuaternión* que representa la rotación del eje x del sistena de coordenadas global hacía el eje x del sistema de coordenadas local del elemento aporticado.

Para esto, se almacena el eje global x y el eje local x en las variables `v_from` y `v_to` respectivamente. El eje global x es igual a $(1,0,0)$ mientras que el eje local x se calcula mediante el método `get_direction_cosines()` (véase el algoritmo 2.12).

Después, se verifica si las variables `v_from` y `v_to` son iguales entre sí, o si una variable es el inverso aditivo de la otra.

En el caso que las variables sean iguales entre sí, no hay rotación y el ángulo θ es igual a cero, por lo tanto el cuaternión es igual a $(1, 0 \times \mathbf{n})$ (véase la ecuación 2-5). En caso contrario, el ángulo θ que describe la rotación es igual a 180°. Como eje se asume el eje global z, por lo que el cuaternión es igual a $(0, 1 \times (0,0,1))$.

Si las variables `v_from` y `v_to` no son iguales entre sí, y una no es el inverso aditivo de la otra, entonces se calcula el eje y el ángulo que describen la rotación del eje global x hacia el eje local x del elemento aporticado, aplicando las expresiones de ecuación (2-4).

Para calcular el eje se halla el producto cruz entre el eje global x y el eje local x mediante la función `cross()`. Después se normaliza el resultado dividiendolo por su norma, con ayuda de la función `linalg.norm()`.

El ángulo se halla calculando el arcocoseno del producto punto entre el eje global x y el eje local x. Esto se calcula mediante las funciones `dot()` y `arccos()` respectivamente.

Finalmente, se aplican las expresiones de la ecuación (2-5) para crear un objeto tipo `Rotation`, mediante la función `Rotation.from_quat()`.

Según Virtanen et al, 2020, la función `from_quat()` permite crear objetos tipo `Rotation`, los cuales son una interfaz para inicializar y representar rotaciones en el espacio, mediante un cuaternión.

Algoritmo 2.13: Método `get_rotation()` de la clase `Frame`.

```
def get_rotation(self):
    """Get rotation"""
    v_from = np.array([1, 0, 0])
    v_to = self.get_direction_cosines()

    if np.all(v_from == v_to):
        return Rotation.from_quat([0, 0, 0, 1])

    elif np.all(v_from == -v_to):
        return Rotation.from_quat([0, 0, 1, 0])

    else:
        w = np.cross(v_from, v_to)
        w = w / linalg.norm(w)
        theta = np.arccos(np.dot(v_from, v_to))

        return Rotation.from_quat([x * np.sin(theta/2) for x in w] + [np.cos(
   theta/2)])
```

get_rotation_matrix()

El método `get_rotation_matrix()` de la clase `Frame` permite calcular la matriz de transformación de rotación de los elementos aporticados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas global.

Según Weaver y Gere, 1990, la matriz de transformación de rotación R_T para un elemento aporticado es

$$\mathbf{R_T} = \begin{bmatrix} \mathbf{R} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{R} \end{bmatrix} \tag{2-7}$$

donde $\mathbf{R}$ es la matriz de rotación presentada en (2-6).

En el algoritmo 2.14 se presenta la implementación del método `get_rotation_matrix()`. El método recibe como argumentos de entrada un *array* que indica para cada grado de libertad si está o no activado. Según los grados de libertad activados se genera la matriz de transformación de rotación de los elementos aporticados.

La matriz de transformación de rotación se genera a partir de la matriz de rotación del elemento aporticado, calculada con el método `get_rotation().as_dcm()` de la clase `Frame` (véase el algoritmo 2.13), y la función `bsr_matrix()`.

Según Virtanen et al, 2020, el método `as_dcm()` de la clase `Rotation` calcula la matriz de rotación de los objetos tipo `Rotation` y la función `bsr_matrix()` crea matrices dispersas con submatrices densas describiendolas en la representación estándar `(data, indices, indptr)`. En dicha representación los índices de la columna de cada submatriz en la fila `i` de la matriz dispersa están almacenados en `indices[indptr[i]:indptr[i+1]]` y los valores correspondientes almacenados en `data[indptr[i]:indptr[i+1]]`.

En las variable `indptr` e `indices` se almacenan los *arrays* `[0, 1, 2]` y `[0, 1]` respectivamente. Estas variables describen en la representación estándar las posiciones que ocupan dos submatrices en la diagonal principal de una matriz dispersa.

En la primer fila hay una submatriz en la primer columna (`indices[indptr[0]:indptr[1]] ->indices[0:1]->indices[0]->0`) y en la segunda fila hay una submatriz en la segunda columna (`indices[indptr[1]:indptr[2]]->indices[1:2]->indices[1]->1`).

Inicialmente se calcula la matriz de transformación de rotación para un solo nodo. Después se seleccionan las filas y las columnas de esta matriz asociadas a los grados de libertad activados. Finalmente se crea toda la matriz de transformación de rotación duplicando los valores seleccionados.

Para crear la matriz de transformación de rotacion para un solo nodo se duplica la matriz de rotación del elemento aporticado, mediante la función `tile`, y se almacena en la variable `data`. Después se pasa junto con las variables `indptr` e `indices` a la función `bsr_matrix()`.

La matriz de transformación de rotación del elemento aporticado se crea al indicar dos matrices de transformación de rotación para un solo nodo en la diagonal principal de una matriz dispersa, después de haber seleccionado las filas y columnas asociadas a los grados de libertad activados. El tamaño de la matriz de transformación de rotación se calcula en función de la cantidad de grados de libertad activados.

Algoritmo 2.14: Método `get_rotation_matrix()` de la clase `Frame`.

```python
def get_rotation_matrix(self, flag_active_joint_displacements):
    """
    Get rotation matrix

    Parameters
    ----------
    flag_active_joint_displacements : array
        Flags active joint's displacements
    """
    # rotation as direction cosine matrix
    indptr = np.array([0, 1, 2])
    indices = np.array([0, 1])
    data = np.tile(self.get_rotation().as_dcm(), (2, 1, 1))

    # matrix rotation for a joint
    t1 = bsr_matrix((data, indices, indptr), shape=(6, 6)).toarray()

    flag_active_joint_displacements = np.nonzero(
   flag_active_joint_displacements)[0]
    n = 2 * np.size(flag_active_joint_displacements)

    t1 = t1[flag_active_joint_displacements[:, None],
   flag_active_joint_displacements]
    data = np.tile(t1, (2, 1, 1))

    return bsr_matrix((data, indices, indptr), shape=(n, n)).toarray()
```

get_local_stiffness_matrix()

El método `get_local_stiffness_matrix()` de la clase `Frame` permite calcular la matriz de rigidez de los elementos aporticados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas local.

Según Weaver y Gere, 1990, (2-8) es la matrix de rigidez del elemento aporticado en coordenadas locales, donde E es el módulo de Young y G es el módulo de elasticidad a cortante del material, L es la longitud del elemento y A_x, I_x, I_y e I_z son el área, la constante de torsión y los momentos principales de inercia de la sección transversal.

$$
\begin{array}{cc}
 & \begin{array}{cccccccccccc} \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} & \mathbf{7} & \mathbf{8} & \mathbf{9} & \mathbf{10} & \mathbf{11} & \mathbf{12} \end{array} \\
\begin{array}{c} \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \\ \mathbf{6} \\ \mathbf{7} \\ \mathbf{8} \\ \mathbf{9} \\ \mathbf{10} \\ \mathbf{11} \\ \mathbf{12} \end{array} &
\begin{bmatrix}
\frac{EA_x}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{EA_x}{L} & 0 & 0 & 0 & 0 & 0 \\
 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} & 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} \\
 & & \frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 & 0 & 0 & -\frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 \\
 & & & \frac{GI_x}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{GI_x}{L} & 0 & 0 \\
 & & & & \frac{4EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 \\
 & & & & & \frac{4EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} \\
 & & & & & & \frac{EA_x}{L} & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & \frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} \\
 & & & & & & & & \frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 \\
 & & & & & & & & & \frac{GI_x}{L} & 0 & 0 \\
 & & & & & & & & & & \frac{4EI_y}{L} & 0 \\
sim. & & & & & & & & & & & \frac{4EI_z}{L}
\end{bmatrix}
\end{array}
\tag{2-8}
$$

En el algoritmo 2.15 se presenta la implementación del método `get_local_stiffness_matrix()`. El método recibe como argumentos de entrada un *array* que indica para cada grado de libertad si está o no activado. Según los grados de libertad activados se genera la matriz de rigidez en el sistema de coordenadas local.

La matriz de rigidez en el sistema de coordenadas local se calcula con los atributos del material, de la sección tranversal, de los nodos de los elementos aporticados y la función `coo_matrix()`.

Según Virtanen et al, 2020, con la función `coo_matrix()` se pueden crear matrices dispersas en el formato coordenado, también conocido como el formato *ijv* o el formato triple. En este formato los indices de las filas, de las columnas y los respectivos valores de la matriz dispersa son almacenados en tres *arrays* independientes `i`, `j` y `data` de tal manera que se cumpla `A[i[k], j[k]] = data[k]`.

En las variables `length`, `e`, `iy` y `iz` se almacenan la longitud del elemento aporticado (véase el algoritmo 2.11), el módulo de Young del material y las inercias principales de la sección transversal con respecto a los ejes y y z del sistema de coordenadas local.

Después se calcula el módulo de Young dividido entre varias potencias de la longitud del elemento aporticado y los resultados se almacena en las variables `el`, `el2` y `el3` respectivamente. El número al final del nombre de estas variables indica la potencia de la longitud del elemento.

Con estas variables se calculan los términos EA/L, GI_x/L, EI_y/L, EI_z/L, $6EI_y/L^2$, $6EI_z/L^2$, $12EI_y/L^3$ y $12EI_z/L^3$, los cuales son almacenados en las variables `ael`, `gjl`, `e_iy_l`, `e_iz_l`, `e_iy_l2`, `e_iz_l2`, `e_iy_l3` y `e_iz_l3`, respectivamente.

Con estas variables se describe la matriz de rigidez en coordenadas locales como una matriz dispersa en el formato *ijv*. Los indices de las filas y las columnas se almacenan en los *arrays* `rows` y `cols`, respectivamente, mientras que los valores de la matriz se almacenan en el *array* `data`.

Por ejemplo, para describir los términos de la matriz de rigidez asociados a las solicitaciones axiales, se le pasa a los *arrays* `rows` y `cols` los valores `[0, 6, 0, 6]` y `[0, 6, 6, 0]`, y al *array* `data` se le pasa los valores `[ael, ael, -eal, -eal]`. Para los otros términos de la matriz de rigidez se procede de manera similar.

Finalmente, se genera la matriz de rigidez del elemento aporticado en el sistema de coordenadas local y se seleccionan las filas y columnas asociadas a los grados de libertad activados.

Algoritmo 2.15: Método `get_local_stiffness_matrix()` de la clase `Frame`.

```
def get_local_stiffness_matrix(self, active_joint_displacements):
    """
    Get local stiffness matrix

    Parameters
    ----------
    active_joint_displacements : array
        Flags active joint's displacements
    """
    length = self.get_length()

    e = self.material.E

    iy = self.section.Iy
    iz = self.section.Iz

    el = e / length
    el2 = e / length ** 2
    el3 = e / length ** 3

    ael = self.section.A * el
    gjl = self.section.Ix * self.material.G / length

    e_iy_l = iy * el
    e_iz_l = iz * el

    e_iy_l2 = 6 * iy * el2
    e_iz_l2 = 6 * iz * el2

    e_iy_l3 = 12 * iy * el3
    e_iz_l3 = 12 * iz * el3
```

```
rows = np.empty(40, dtype=int)
cols = np.empty(40, dtype=int)
data = np.empty(40)

# AE / L
rows[:4] = np.array([0, 6, 0, 6])
cols[:4] = np.array([0, 6, 6, 0])
data[:4] = np.array([ael, ael, -ael, -ael])

# GJ / L
rows[4:8] = np.array([3, 9, 3, 9])
cols[4:8] = np.array([3, 9, 9, 3])
data[4:8] = np.array([gjl, gjl, -gjl, -gjl])

# 12EI / L^3
rows[8:12] = np.array([1, 7, 1, 7])
cols[8:12] = np.array([1, 7, 7, 1])
data[8:12] = np.array([e_iz_l3, e_iz_l3, -e_iz_l3, -e_iz_l3])

rows[12:16] = np.array([2, 8, 2, 8])
cols[12:16] = np.array([2, 8, 8, 2])
data[12:16] = np.array([e_iy_l3, e_iy_l3, -e_iy_l3, -e_iy_l3])

# 6EI / L^2
rows[16:20] = np.array([1, 5, 1, 11])
cols[16:20] = np.array([5, 1, 11, 1])
data[16:20] = np.array([e_iz_l2, e_iz_l2, e_iz_l2, e_iz_l2])

rows[20:24] = np.array([5, 7, 7, 11])
cols[20:24] = np.array([7, 5, 11, 7])
data[20:24] = np.array([-e_iz_l2, -e_iz_l2, -e_iz_l2, -e_iz_l2])

rows[24:28] = np.array([2, 4, 2, 10])
cols[24:28] = np.array([4, 2, 10, 2])
data[24:28] = np.array([-e_iy_l2, -e_iy_l2, -e_iy_l2, -e_iy_l2])

rows[28:32] = np.array([4, 8, 8, 10])
cols[28:32] = np.array([8, 4, 10, 8])
data[28:32] = np.array([e_iy_l2, e_iy_l2, e_iy_l2, e_iy_l2])

# 4EI / L
rows[32:36] = np.array([4, 10, 5, 11])
cols[32:36] = np.array([4, 10, 5, 11])
data[32:36] = np.array([4 * e_iy_l, 4 * e_iy_l, 4 * e_iz_l, 4 * e_iz_l])

rows[36:] = np.array([10, 4, 11, 5])
cols[36:] = np.array([4, 10, 5, 11])
```

```
    data[36:] = np.array([2 * e_iy_l, 2 * e_iy_l, 2 * e_iz_l, 2 * e_iz_l])

    k = coo_matrix((data, (rows, cols)), shape=(12, 12)).toarray()

    active_frame_displacement = np.nonzero(np.tile(active_joint_displacements,
    2))[0]

    return k[active_frame_displacement[:, None], active_frame_displacement]
```

get_global_stiffness_matrix()

El método `get_global_stiffness_matrix()` de la clase `Frame` permite calcular la matriz de rigidez de los elementos aporticados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas global.

Según Weaver y Gere, 1990, la matriz de rigidez de los elementos aporticados con respecto al sistema de coordenadas global se puede calcular como

$$\mathbf{S_{MS}} = \mathbf{R_T} \mathbf{S_M} \mathbf{R_T^T} \tag{2-9}$$

donde $\mathbf{R_T}$ y $\mathbf{S_M}$ son la matriz de transformación de rotación y la matriz de rigidez en el sistema de coordenadas local del elemento aporticado.

En el algoritmo 2.16 se presenta la implementación del método `get_global_stiffness_matrix()`. El método recibe como argumento de entrada un *array* que indica para cada grado de libertad si está o no activado. Según los grados de libertad activados se genera la matriz de rigidez en el sistema de coordenadas global.

La matriz de rigidez en el sistema de coordenadas global se calcula con la matriz de rigidez en el sistema de coordenadas local y la matriz de transformación de rotación del elemento aporticado. Estas matrices son calculadas con los métodos `get_matrix_rotation()` (véase el algoritmo 2.14) y `get_local_stiffness_matrix()` (véase el algoritmo 2.15), y almacenadas en las variables `k` y `t`, respectivamente.

Finalmente, se operan las matrices obtenidas según (2-9) para calcular la matriz de rigidez del elemento aporticado en el sistema de coordenadas global con las funciones `dot()` y `transpose()`.

Algoritmo 2.16: Método `get_global_stiffness_matrix()` de la clase `Frame`.

```
def get_global_stiffness_matrix(self, active_joint_displacements):
    """
    Get the global stiffness matrix
```

```
    Parameters
    ----------
    active_joint_displacements : array
        Flags active joint's displacements
    """
    k = self.get_local_stiffness_matrix(active_joint_displacements)
    t = self.get_rotation_matrix(active_joint_displacements)

    return np.dot(np.dot(t, k), np.transpose(t))
```

2.1.6. Support

La clase `Support` representa los apoyos de la estructura, al establecer los desplazamientos restringidos de los nodos.

En el algortimo 2.17 se presenta la implementación de la clase `Support`. Como mecánismo de optimización, se asigna una tupla con los elementos `'ux'`, `'uy'`, `'uz'`, `'rx'`, `'ry'` y `'rz'` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia.

El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los grados de libertad, los cuales tienen `False` como valor por defecto. El usuario debe indicar qué grados de libertad están restrigidos.

Finalmente, la clase `Support` implementa el método `get_restrains()` que genera un *array* que indica para cada grado de libertad activado si está o no restrigido.

Algoritmo 2.17: Clase `Support` implementada en el archivo `primitives.py`.

```
class Support(AttrDisplay):
    """
    Point of support

    Attributes
    ----------
    ux : bool
        Flag restrain x-axis translation.
    uy : bool
        Flag restrain y-axis translation.
    uz : bool
        Flag restrain z-axis translation.
    rx : bool
        Flag restrain x-axis rotation.
```

```
    ry : bool
        Flag restrain y-axis rotation.
    rz : bool
        Flag restrain z-axis rotation.

    Methods
    -------
    get_restrains()
        Get flag restrains.
    """

    __slots__ = ('ux', 'uy', 'uz', 'rx', 'ry', 'rz')

    def __init__(self, ux=False, uy=False, uz=False, rx=False, ry=False, rz=
False):
        """
        Instantiate a Support object

        Parameters
        ----------
        ux : bool
            Flag restrain x-axis translation.
        uy : bool
            Flag restrain y-axis translation.
        uz : bool
            Flag restrain z-axis translation.
        rx : bool
            Flag restrain x-axis rotation.
        ry : bool
            Flag restrain y-axis rotation.
        rz : bool
            Flag restrain z-axis rotation.
        """
        self.ux = ux
        self.uy = uy
        self.uz = uz
        self.rx = rx
        self.ry = ry
        self.rz = rz

    def get_restrains(self, flag_joint_displacements):
        """
        Get restrains

        Attributes
        ----------
        flag_joint_displacements : array
            Flag active joint displacements.
```

```
        """
        return np.array([getattr(self, name) for name in self.__slots__])[
    flag_joint_displacements]
```

2.1.7. LoadPattern

La clase `LoadPattern` representa los patrones de carga a los que está sometida la estructura, al establecer la magnitud de las fuerzas y las cargas distribuidas que actúan en los nodos y en los elementos aporticados, respectivamente. En la figura **2-4** se presentan los métodos y atributos de esta clase.

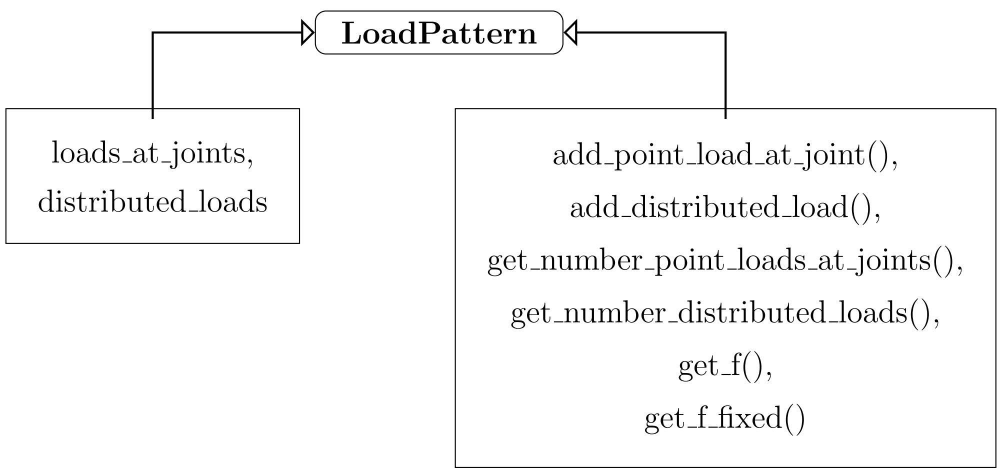

Figura 2-4: Métodos y atributos de la clase `LoadPattern`.

Como mecánismo de optimización, se asigna una tupla con los elementos `loads_at_joints` y `distributed_loads` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia.

En el algoritmo 2.18 se presenta el constructor de la clase `LoadPattern`. El constructor de la clase no tiene argumentos de entrada. Sin embargo, asigna un diccionario vacío a los atributos `loads_at_joints` y `distributed_loads`.

Algoritmo 2.18: Constructor de la clase `LoadPattern`.

```
def __init__(self):
    """Instantiate a LoadPatter object"""
    self.loads_at_joints = {}
    self.distributed_loads = {}
```

A continuación se presentan los métodos de la clase `LoadPattern`, con los cuales se puede, entre otras cosas, calcular el vector de fuerzas en los nodos de la estructura del caso de carga.

add_point_load_at_joint()

El método `add_point_load_at_joint()` de la clase `LoadPattern` permite agregar fuerzas en los nodos.

En el algoritmo 2.19 se presenta la implementación del método `add_point_load_at_joint()`. Los argumentos de entrada opcionales `*args` y `**kwargs` son pasados al constructor de la clase `PointLoad`, mientras que el argumento `joint` es usado como llave para almacenar el objeto creado en el diccionario `loads_at_joints`.

Algoritmo 2.19: Método `add_point_load_at_joint()` de la clase `LoadPattern`.

```
def add_point_load_at_joint(self, joint, *args, **kwargs):
    """
    Add a point load at joint

    Parameters
    ----------
    joint : Joint
        Joint.
    """
    self.loads_at_joints[joint] = PointLoad(*args, **kwargs)
```

add_distributed_load()

El método `add_distributed_load()` de la clase `LoadPattern` permite agregar cargas distribuidas en los elementos aporticados.

En el algoritmo 2.20 se presenta la implementación del método `add_distributed_load()`. Los argumentos de entrada opcionales `*args` y `**kwargs` son pasados al constructor de la clase `DistributedLoad`, mientras que el argumento `frame` es usado como llave para almacenar el objeto creado en el diccionario `distributed_loads`.

Algoritmo 2.20: Método `add_distributed_load()` de la clase `LoadPattern`.

```
def add_distributed_load(self, frame, *args, **kwargs):
    """
    Add a distributed load at frame
    Parameters
```

```
    ___________
    frame : Joint
        Frame
    """
    self.distributed_loads[frame] = DistributedLoad(*args, **kwargs)
```

get_number_point_loads_at_joints()

El método `get_number_point_loads_at_joints()` de la clase `LoadPattern` calcula el número de nodos cargados.

En el algoritmo 2.21 se presenta la implementación del método `get_number_point_loads_at_joints()`. El método calcula la cantidad de entradas que tiene el diccionario `loads_at_joints`.

Algoritmo 2.21: Método `get_number_point_loads_at_joints()` de la clase `LoadPattern`.

```python
def get_number_point_loads_at_joints(self):
    """Get number loads at joints"""
    return len(self.loads_at_joints)
```

get_number_distributed_loads()

El método `get_number_distributed_loads()` de la clase `LoadPattern` calcula el número de elementos aporticados cargados.

En el algoritmo 2.22 se presenta la implementación del método `get_number_distributed_loads()`. El método calcula la cantidad de entradas que tiene el diccionario `distributed_loads`.

Algoritmo 2.22: Método `get_number_distributed_loads()` de la clase `LoadPattern`.

```python
def get_number_distributed_loads(self):
    """Get number distributed loads"""
    return len(self.distributed_loads)
```

get_f()

El método `get_f()` de la clase `LoadPattern` calcula el vector de fuerzas total en los nodos de la estructura del caso de carga, representado por objetos tipo `LoadPattern`, con respecto al sistema de coordenadas global.

Según Weaver y Gere, 1990, el vector de fuerzas equivalente en los nodos de la estructura A_E debido a las cargas en los elementos aporticados se calcula como

$$\mathbf{A_E} = -\sum_{i=1}^{m} \mathbf{A_{MSi}} \tag{2-10}$$

donde A_{MSi} es el vector de acciones fijas en los nodos del elemento aporticado i en el sistema de coordenadas global. Este vector de fuerzas equivalentes se suma con el vector de fuerzas aplicadas en los nodos de la estructura para formar el vector de fuerzas total.

En el algoritmo 2.23 se presenta la implementación del método `get_f()`. El método recibe los argumentos de entrada obligatorios `flag_displacements` e `indexes`. La variable `flag_displacements` indica para cada grado de libertad si está o no activado, mientras que la variable `indexes` relaciona los objetos tipo `Joint` con sus respectivos grados de libertad. Según los grados de libertad activados se genera el vector de fuerzas total en los nodos de la estructura del caso de carga.

El vector de fuerzas aplicadas en los nodos de la estructura se ensambla, a partir de las fuerzas aplicadas en cada nodo de la estructura y sus respectivos grados de libertad, con la función `coo_matrix()`.

Para esto, primero se calcula la cantidad de grados de libertad activados y nodos cargados, con la función `count_nonzero` y el método `get_number_point_loads_at_joints()` (véase el algoritimo 2.21), y se almacenan en las variables `no` y `n`, respectivamente.

Con estos valores se dimensionan los *arrays* `rows`, `cols` y `data`. Los *arrays* `rows` y `data` se crean con valores arbitrarios, con la función `np.zeros()`, para almacenar los grados de libertad y las fuerzas en los nodos respectivamente, mientras que el *array* `cols` se crea con ceros en todas sus entradas, con la función `np.zeros()`, debido a que el vector de fuerzas en los nodos de la estructura es un vector columna.

Finalmente, se crea el vector de fuerzas del caso de carga en los nodos de la estructura, pasando a la función `coo_matrix()` los *arrays* `rows`, `cols` y `data`, y se le resta el vector de fuerzas equivalentes del caso de carga en los nodos, calculada con el método `get_f_fixed()`.

Algoritmo 2.23: Método `get_f()` de la clase `LoadPattern`.

```
def get_f(self, flag_displacements, indexes):
    """
    Get the load vector

    Attributes
```

```
    ___________
    flag_displacements : array
        Flags active joint's displacements
    indexes : dict
        Key value pairs joints and indexes.
    """
    no = np.count_nonzero(flag_displacements)

    n = self.get_number_point_loads_at_joints()

    rows = np.empty(n * no, dtype=int)
    cols = np.zeros(n * no, dtype=int)
    data = np.empty(n * no)

    for i, (joint, point_load) in enumerate(self.loads_at_joints.items()):
        rows[i * no:(i + 1) * no] = indexes[joint]
        data[i * no:(i + 1) * no] = point_load.get_load(flag_displacements)

    return coo_matrix((data, (rows, cols)), (no * len(indexes), 1)) - self.
     get_f_fixed(flag_displacements, indexes)
```

get_f_fixed()

El método `get_f_fixed()` de la clase `LoadPattern` calcula el vector de fuerzas equivalente en los nodos de la estructura del caso de carga, representado por objetos tipo `LoadPattern`, con respecto al sistema de coordenadas global.

En el algoritmo 2.24 se presenta la implementación del método `get_f_fixed()`. El método recibe los argumentos de entrada obligatorios `flag_displacements` e `indexes`. La variable `flag_displacements` indica para cada grado de libertad si está o no activado, mientras que la variable `indexes` relaciona los objetos tipo `Joint` con sus respectivos grados de libertad. Según los grados de libertad activados se genera el vector de fuerzas equivalentes en los nodos de la estructura del caso de carga.

El vector de fuerzas equivalentes en los nodos de la estructura se ensambla, a partir de las cargas aplicadas en los elementos aporticados y sus respectivos grados de libertad, con la función `coo_matrix()`.

Para esto, primero se calcula la cantidad de grados de libertad activados y elementos aporticados cargados, con la función `count_nonzero` y el método `get_number_distributed_loads()` (véase el algoritmo 2.22), y se almacenan en las variables `no` y `n`, respectivamente.

Con estos valores se dimensionan los *arrays* `rows`, `cols` y `data`. Los *arrays* `rows` y `data` se crean con valores arbitrarios, para almacenar los grados de libertad y las fuerzas en los nodos respectivamente, mientras que el *array* `cols` se crea con ceros entodas sus entradas, debido a que el vector de fuerzas equivalente en los nodos de la estructura es un vector columna.

Algoritmo 2.24: Método `get_f_fixed()` de la clase `LoadPattern`.

```
def get_f_fixed(self, flag_joint_displacements, indexes):
    """
    Get the f fixed.

    Attributes
    ----------
    flag_joint_displacements : array
        Flags active joint's displacements.
    indexes : dict
        Key value pairs joints and indexes.
    """
    no = np.count_nonzero(flag_joint_displacements)

    n = self.get_number_distributed_loads()

    rows = np.empty(2 * n * no, dtype=int)
    cols = np.zeros(2 * n * no, dtype=int)
    data = np.empty(2 * n * no)

    for i, (frame, distributed_load) in enumerate(self.distributed_loads.items()
    ):
        joint_j = frame.joint_j
        joint_k = frame.joint_k

        rows[i * 2 * no:(i + 1) * 2 * no] = np.concatenate((indexes[joint_j],
    indexes[joint_k]))
        data[i * 2 * no:(i + 1) * 2 * no] = distributed_load.get_f_fixed(
    flag_joint_displacements, frame)

    return coo_matrix((data, (rows, cols)), (no * len(indexes), 1))
```

2.1.8. PointLoad

La clase `PointLoad` representa las fuerzas aplicadas en los nodos de la estructura, al establecer el valor de las fuerzas en el sistema de coordenadas global.

En el algoritmo 2.25 se presenta la implementación de la clase `PointLoad`. Como mecánismo de optimización, se asigna una tupla con los elementos `'fx'`, `'fy'`, `'fz'`, `'mx'`, `'my'` y `'mz'` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia.

El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los grados de libertad, los cuales tienen 0 como valor por defecto. El usuario debe indicar el valor de las fuerzas diferentes de cero.

Finalmente, la clase `PointLoad` implementa el método `get_load()` que genera un *array* que indica, para cada grado de libertad activado, el valor de la fuerza.

Algoritmo 2.25: Clase `PointLoad` implementada en el archivo `primitives.py`.

```
class PointLoad(AttrDisplay):
    """
    Point load

    Attributes
    ----------
    fx : float
        Force along 'x' axis.
    fy : float
        Force along 'y' axis.
    fz : float
        Force along 'z'axis.
    mx : float
        Force around 'x' axis.
    my : float
        Force around 'y' axis.
    mz : float
        Force around 'z' axis.

    Methods
    -------
     get_load(flag_joint_displacements)
        Get the load vector.
    """
    __slots__ = ('fx', 'fy', 'fz', 'mx', 'my', 'mz')

    def __init__(self, fx=0, fy=0, fz=0, mx=0, my=0, mz=0):
        """
        Instantiate a PointLoad object

        Parameters
        ----------
        fx : float
            Force along 'x' axis.
        fy : float
            Force along 'y' axis.
        fz : float
```

```
            Force along 'z' axis.
        mx : float
            Force around 'x' axis.
        my : float
            Force around 'y' axis.
        mz : float
            Force around 'z' axis.
        """
        self.fx = fx
        self.fy = fy
        self.fz = fz

        self.mx = mx
        self.my = my
        self.mz = mz

    def get_load(self, flag_joint_displacements):
        """
        Get load

        Parameters
        ----------
        flag_joint_displacements : array
            Flags active joint's displacements.
        """

        return np.array([getattr(self, name) for name in self.__slots__])[
    flag_joint_displacements]
```

2.1.9. DistributedLoad

La clase `DistributedLoad` representa las cargas distribuidas aplicadas en los elementos aporticados de la estructura, al establecer el valor de las cargas en el sistema de coordenadas local.

En el algoritmo 2.26 se presenta la implementación de la clase `DistributedLoad`. Como mecánismo de optimización, se asigna una tupla con los elementos `'system'`, `'fx'`, `'fy'` y `'fz'` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instacia.

El constructor de la clase recibe tres argumentos de entrada opcionales, para cada una de las cargas distribuidas a lo largo de los ejes principales del sistema de coordenadas local. El usuario debe indicar el valor de las cargas distribuidas diferentes de cero.

Finalmente, la clase `DistributedLoad` implementa el método `get_f_fixed()` que genera un *array* que indica, para cada grado de libertad activado, las fuerzas equivalentes en los nodos de la estructura en el sistema de coordenadas global.

Algoritmo 2.26: Clase `DistributedLoad` implementada en el archivo `primitives.py`.

```python
class DistributedLoad(AttrDisplay):
    """
    Distributed load

    Attributes
    ----------
    system: str
        Coordinate system ('local' by default).
    fx : float
        Distributed force along 'x' axis.
    fy : float
        Distributed force along 'y' axis.
    fz : float
        Distributed force along 'z' axis.

    Methods
    -------
    get_load()
        Get the load vector.
    """
    __slots__ = ('system', 'fx', 'fy', 'fz')

    def __init__(self, fx=0, fy=0, fz=0):
        """
        Instantiate a Distributed object

        Parameters
        ----------
        fx : float
            Distributed force along 'x' axis.
        fy : float
            Distributed force along 'y' axis.
        fz : float
            Distributed force along 'z' axis.
        """
        self.system = 'local'

        self.fx = fx
        self.fy = fy
        self.fz = fz

    def get_f_fixed(self, flag_joint_displacements, frame):
        """
```

```
        Get f fixed.

        Parameters
        ----------
        flag_joint_displacements : array
            Flags active joint's displacements.
        frame : Frame
            Frame.
        """
        length = frame.get_length()

        fx = self.fx
        fy = self.fy
        fz = self.fz

        f_local = [-fx * length / 2, -fy * length / 2, -fz * length / 2, 0, fz
 * length ** 2 / 12, -fy * length ** 2 / 12]
        f_local += [fx * length / 2, -fy * length / 2, -fz * length / 2, 0, -
fz * length ** 2 / 12, fy * length ** 2 / 12]

        return np.dot(frame.get_rotation_matrix(flag_joint_displacements),
f_local)
```

2.1.10. Displacement

La clase `Displacement` representa los desplazamientos de los nodos de la estructura, al establecer el valor de las translaciones y rotaciones en el sistema de coordenadas global.

En el algoritmo 2.27 se presenta la implementación de la clase `Displacement`. Como mecánismo de optimización, se asigna una tupla con los elementos `'ux'`, `'uy'`, `'uz'`, `'rx'`, `'ry'` y `'rz'` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia.

El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los posibles desplazamientos de los nodos en el sistema de coordenadas global. El usuario debe indicar el valor de los desplazamientos diferentes de cero.

Finalmente, la clase `Displacement` implementa el método `get_displacements()` que genera un *array* que indica, para cada grado de libertad activado, el valor del desplazamiento.

Algoritmo 2.27: Clase `Displacement` implementada en el archivo `primitives.py`.

```
class Displacement(AttrDisplay):
    """
```

```
    Displacement

    Attributes
    ----------
    ux : float
        Translation along 'x' axis.
    uy : float
        Translation along 'y' axis.
    uz : float
        Translation along 'z' axis.
    rx : float
        Rotation around 'x' axis.
    ry : float
        Rotation around 'y' axis.
    rz : float
        Rotation around 'z' axis.

    Methods
    -------
    get_displacements()
        Get the displacement vector.
    """
    __slots__ = ('ux', 'uy', 'uz', 'rx', 'ry', 'rz')

    def __init__(self, ux=0, uy=0, uz=0, rx=0, ry=0, rz=0):
        """
        Instantiate a Displacement

        Parameters
        ----------
        ux : float
            Translation along 'x' axis.
        uy : float
            Translation along 'y' axis.
        uz : float
            Translation along 'z' axis.
        rx : float
            Rotation around 'x' axis.
        ry : float
            Rotation around 'y' axis.
        rz : float
            Rotation around 'z' axis.
        """
        self.ux = ux
        self.uy = uy
        self.uz = uz

        self.rx = rx
```

```
        self.ry = ry
        self.rz = rz

    def get_displacement(self, flag_joint_displacements):
        """Get displacements"""
        return np.array([getattr(self, name) for name in self.__slots__])[
    flag_joint_displacements]
```

2.1.11. Reaction

La clase `Reaction` representa las reacciones de los apoyos de la estructura, al establecer el valor de las reacciones en el sistema de coordenadas global.

En el algoritmo 2.28 se presenta la implementación de la clase `Reaction`. Como mecánismo de optimización, se asigna una tupla con los elementos `'fx'`, `'fy'`, `'fz'`, `'mx'`, `'my'` y `'mz'` al atributo `__slots__` de la clase para indicarle a Python que limite la cantidad de atributos que puede tener una instancia.

El constructor de la clase recibe seis argumentos de entrada opcionales, para cada una de las posibles reacciones en el sistema de coordenadas global. El usuario debe indicar el valor de las reacciones diferentes de cero.

Finalmente, la clase `Reaction` implementa el método `get_reactions()` que genera un *array* que indica, para cada grado de libertad activado, el valor de la reacción.

Algoritmo 2.28: Clase `Reaction` implementada en el archivo `primitives.py`.

```
class Reaction(AttrDisplay):
"""
Reaction

Attributes
----------
fx : float
    Force along 'x' axis.
fy : float
    Force along 'y' axis.
fz : float
    Force along 'z' axis.
mx : float
    Moment around 'x' axis.
my : float
    Moment around 'y' axis.
mz : float
```

```
    Moment around 'z' axis.

Methods
-------
get_reactions()
    Get the load vector.
"""
__slots__ = ('fx', 'fy', 'fz', 'mx', 'my', 'mz')

def __init__(self, fx=0, fy=0, fz=0, mx=0, my=0, mz=0):
    """
    Instantiate a Reaction

    Parameters
    ----------
    fx : float
        Force along 'x' axis.
    fy : float
        Force along 'y' axis.
    fz : float
        Force along 'z' axis.
    mx : float
        Moment around 'x' axis.
    my : float
        Moment around 'y' axis.
    mz : float
        Moment around 'z' axis.
    """
    self.fx = fx
    self.fy = fy
    self.fz = fz
    self.mx = mx
    self.my = my
    self.mz = mz

def get_reactions(self, flag_joint_displacements):
    """Get reactions"""
    return np.array([getattr(self, name) for name in self.__slots__])[
   flag_joint_displacements]
```

2.2. Structure

La clase `Structure` representa el modelo de estructuras aporticadas tridimensionales sometidas a cargas estáticas, al agregar objetos que describen la geometría de la estructura, sus condiciones de apoyo y las solicitaciones externas. En la figura **2-5** se presentan los métodos y atributos de esta clase.

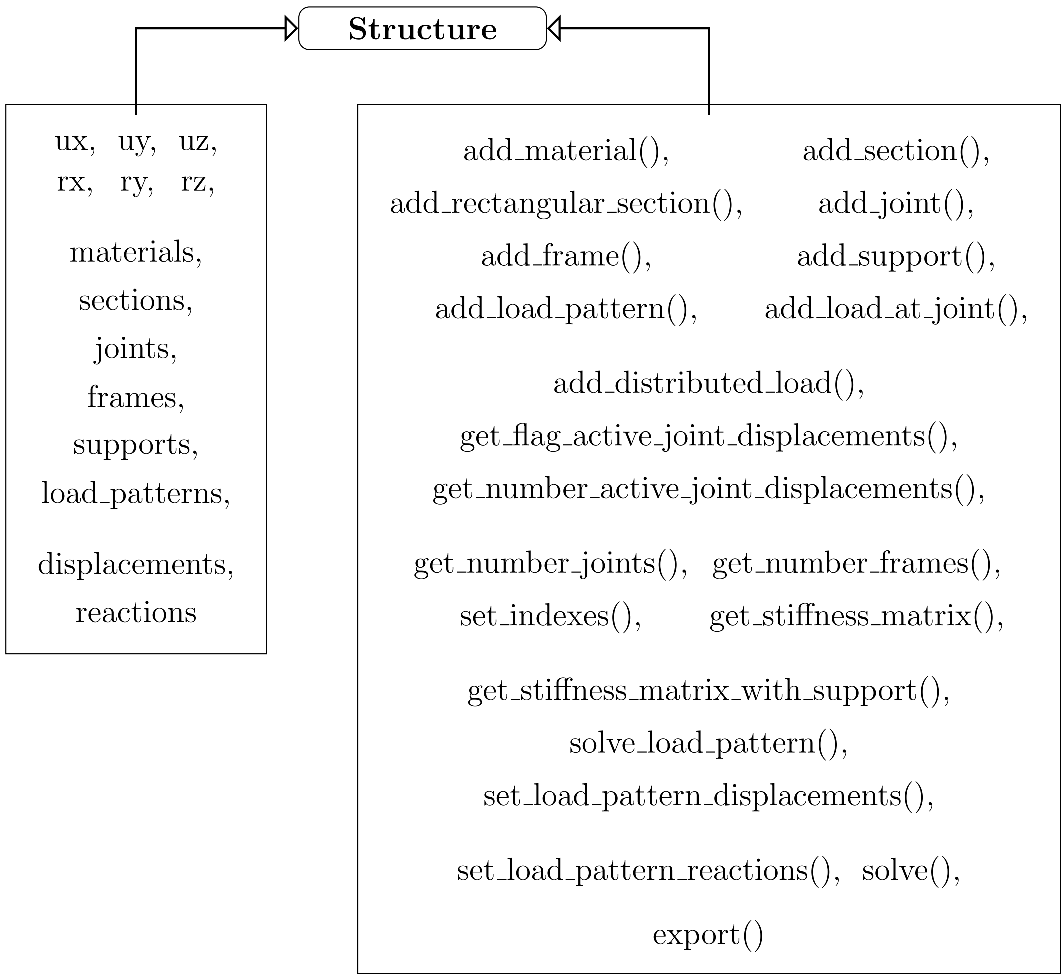

Figura 2-5: Métodos y atributos de la clase `Structure` (repetida).

Como se mencionó anteriormente, el constructor de la clase recibe seis argumentos de entrada opcionales, uno para cada grado de libertad, los cuales tienen `False` como valor por defecto. Cuando el usuario crea un objeto de esta clase debe indicar qué grados de libertad quiere tener en cuenta para analizar el modelo (véase el algoritmo 2.1).

Con los métodos `add_material()`, `add_section()`, `add_rectangular_section()`, `add_joint()`, `add_frame()` y `add_support()` se pueden agregar objetos tipo `Material`, `Section`, `RectangularSection`, `Joint`, `Frame` y `Support`, respectivamente, para describir la geometría y condiciones de apoyo de la estructura.

Con los métodos `add_load_pattern()`, `add_load_at_joint()` y `add_distributed_load()` de pueden agregar objetos tipo `LoadPattern`, `PointLoad` y `DistributedLoad`, respectivamente, para describir las cargas de los patrones de carga a los que se encuentra sometida la estructura.

A continuación se presentan los demás métodos de la clase `Structure`, con los cuales se puede, entre otras cosas, analizar linealmente el modelo para encontrar los desplazamientos y reacciones de la estructura sometida a los diferentes patrones de carga.

2.2.1. get_flag_active_joint_displacements()

El método `get_flag_active_joint_displacements()` de la clase `Structure` genera un *array* que indica para cada grado de libertad si está o no activado.

En el algoritmo 2.29 se presenta la implementación del método `get_flag_active_joint_displacements()`. El método genera un *array* con los valores de los atributos `ux`, `uy`, `uz`, `rx`, `ry` y `rz` como entradas.

Algoritmo 2.29: Método `get_flag_active_joint_displacements()` de la clase `Structure`.

```python
def get_flag_active_joint_displacements(self):
    """
    Get active joint displacements

    Returns
    -------
    indexes: array
        Flag active joint displacements.
    """
    return np.array([self.ux, self.uy, self.uz, self.rx, self.ry, self.rz])
```

2.2.2. get_number_active_joint_displacements()

El método `get_number_active_joint_displacements()` de la clase `Structure` calcula el número de grados de libertad activados.

En el algoritmo 2.30 se presenta la implementación del método `get_number_active_joint_displacements()`. El método calcula la cantidad de entradas iguales a `True` del *array* generado por el método `get_flag_active_joint_displacements()` (véase el algoritmo 2.29).

Algoritmo 2.30: Método `get_number_active_joint_displacements()` de la clase `Structure`.

```python
def get_flag_active_joint_displacements(self):
    """
    Get active joint displacements
```

```
    Returns
    -------
    array
        Flags active joint displacements.
    """
    return np.array([self.ux, self.uy, self.uz, self.rx, self.ry, self.rz])
```

2.2.3. get_number_joints()

El método `get_number_joints()` de la clase `Structure` calcula cantidad de nodos de la estructura.

En el algoritmo 2.31 se presenta la implementación del método `get_number_joints()`. El método calcula la cantidad de entradas que tiene el diccionario `joints`.

Algoritmo 2.31: Método `get_number_joints()` de la clase `Structure`.

```
def get_number_joints(self):
    """Get number of joints

    Returns
    -------
    int
        Number of joints.
    """
    return len(self.joints)
```

2.2.4. get_number_frames()

El método `get_number_frames()` de la clase `Structure` calcula la cantidad de elementos aporticados de la estructura.

En el algoritmo 2.32 se presenta la implementación del métdo `get_number_frames()`. El método calcula la cantidad de entradas que tiene el diccionario `frames`.

Algoritmo 2.32: Método `get_number_frames()` de la clase `Structure`.

```
def get_number_frames(self):
    """Get number of frames

    Returns
    -------
```

```
        int
            Number of frames
        """
        return len(self.frames)
```

2.2.5. set_indexes()

El método `set_indexes()` de la clase `Structure` genera un diccionario donde las llaves son los nodos de la estructura y los valores los respectivos grados de libertad.

En el algoritmo 2.33 se presenta la implementación del método `set_indexes()`. El método crea un diccionario donde las llaves son los objetos tipo `Joint` del diccionario `joints` y los valores *arrays* con los respectivos grados de libertad.

Los grados de libertad de cada nodo de la estructura se asignan de manera secuencial en función de la cantidad de grados de libertad activados, calculada con el método `get_number_active_joint_displacements()` (véase el algoritmo 2.30). Al primer nodo se le asignan los primeros `n` indices, comenzando desde cero, al segundo nodo los siguientes `n` indices y así sucesivamente para cada uno de los demás nodos.

Algoritmo 2.33: Método `set_indexes()` de la clase `Structure`.

```
def set_indexes(self):
    """Set the indexes"""
    n = self.get_number_active_joint_displacements()

    return {joint: np.arange(n * i, n * (i + 1)) for i, joint in enumerate(
    self.joints.values())}
```

2.2.6. get_stiffness_matrix()

El método `get_stiffness_matrix()` de la clase `Structure` permite calcular la matriz de rigidez de la estructura.

En el algoritmo 2.34 se presenta la implementación del método `get_stiffness_matrix()`. El método recibe como argumentos de entrada el diccionario que relaciona los nodos de la estructura con sus respectivos grados de libertad, calculado con el método `set_indexes()` (véase el algoritmo 2.33). Según los grados de libertad de los nodos de la estructura se ensamblan las matrices de rigidez de los elementos aporticados con la función `coo_matrix()`.

Para cada objeto tipo `Frame` del diccionario `frames` se calcula su matriz de rigidez en el sistema de coordenadas global, mediante el método `get_global_stiffness_matrix()` (véase el algoritmo 2.16), y se extraen los grados de libertad de los nodos del elemento aporticado del diccionario `indexes`.

Con estas variables se describe la matriz de rigidez en coordenadas locales como una matriz dispersa en el formato *ijv*. Los indices de las filas y las columnas se almacenan en los *arrays* `rows` y `cols`, respectivamente, mientras que los valores de la matriz se almacenan en el *array* `data`.

Finalmente, se genera la matriz de rigidez de la estructura indicando que se trata de una matriz cuadrada de tamaño del número de grados de libertad activados por la cantidad de nodos de la estructura.

Algoritmo 2.34: Método `get_stiffness_matrix()` de la clase `Structure`.

```python
def get_stiffness_matrix(self, indexes):
    """
    Get the stiffness matrix of the structure

    Parameters
    ----------
    indexes : dict
        Key value pairs joints and indexes.

    Returns
    -------
    k : coo_matrix
        Stiffness matrix of the structure.
    """
    flag_joint_displacements = self.get_flag_active_joint_displacements()
    number_active_joint_displacements = np.count_nonzero(
   flag_joint_displacements)

    number_joints = self.get_number_joints()
    number_frames = self.get_number_frames()

    # just for elements with two joints
    n = 2 * number_active_joint_displacements  # change function element type
    n_2 = n ** 2

    rows = np.empty(number_frames * n_2, dtype=int)
    cols = np.empty(number_frames * n_2, dtype=int)
    data = np.empty(number_frames * n_2)
```

```
    for i, frame in enumerate(self.frames.values()):
        k_element = frame.get_global_stiffness_matrix(flag_joint_displacements
)
        indexes_element = np.concatenate((indexes[frame.joint_j], indexes[
frame.joint_k]))
        indexes_element = np.broadcast_to(indexes_element, (n, n))

        rows[i * n_2:(i + 1) * n_2] = indexes_element.flatten('F')
        cols[i * n_2:(i + 1) * n_2] = indexes_element.flatten()
        data[i * n_2:(i + 1) * n_2] = k_element.flatten()

    return coo_matrix((data, (rows, cols)), 2 * (
number_active_joint_displacements * number_joints,))
```

2.2.7. get_stiffness_matrix_with_support()

El método `get_stiffness_matrix_with_support()` de la clase `Structure` modifica la matriz de rigidez de la estructura, calculada con el método `get_stiffness_matrix()` (véase el algoritmo 2.34), para tener en cuenta las condiciones de apoyo.

Según Reddy, 1993, para tener en cuenta las condiciones de apoyo de la estructura en la matriz de rigidez, se deben reemplazar los valores de las filas y las columnas asociadas a los grados de libertad restringidos por ceros, a excepción de los valores en la diagonal principal, los cuales deben ser reemplazados por 1.

En el algoritmo 2.35 se presenta la implementación del método `get_stiffness_matrix_with_support()`. El método recibe como argumentos de entrada la matriz de rigidez de la estructura, calculada con el método `get_stiffness_matrix()` (véase el algoritmo 2.34), y el diccionario que relaciona los nodos de la estructura con sus respectivos grados de libertad, calculado con el método `set_indexes()` (véase el algoritmo 2.33).

Para cada objeto tipo `Support` del diccionario `supports` se extraen los grados de libertad del diccionario `indexes` y se calculan las restricciones del apoyo con el método `get_restrains()` (véase el algoritmo 2.17). Estos valores se almacenan en las variables `joint_indexes` y `restrains` respectivamente.

Finalmente, para cada grado de libertad restringido se reemplazan los valores asociados de la fila y la columna de la matriz de rigidez de la estructura por ceros y el valor en la diagonal principal por 1.

Algoritmo 2.35: Método `get_stiffness_matrix_with_support()` de la clase `Structure`.

```
def get_stiffness_matrix_with_support(self, stiffness_matrix, indexes):
    """
    Get the stiffness matrix of the structure with supports

    Parameters
    ----------
    stiffness_matrix : ndarray
        Stiffness matrix of the structure.
    indexes : dict
        Key value pairs joints and indexes.

    Returns
    -------
    stiffness_matrix_with_supports : ndarray
        Stiffness matrix of the structure modified by supports.
    """
    flag_joint_displacements = self.get_flag_active_joint_displacements()
    n = np.shape(stiffness_matrix)[0]

    for joint, support in self.supports.items():
        joint_indexes = indexes[joint]
        restrains = support.get_restrains(flag_joint_displacements)

        for index in joint_indexes[restrains]:
            stiffness_matrix[index] = stiffness_matrix[:, index] = np.zeros(n)
            stiffness_matrix[index, index] = 1

    return stiffness_matrix
```

2.2.8. solve_load_pattern()

El método `solve_load_pattern()` de la clase `Structure` calcula los vectores de desplazamientos y fuerzas en los nodos de la estructura debidos a las cargas definidas en los patrones de carga.

En el algoritmo 2.36 se presenta la implementación del método `solve_load_pattern()`. El método recibe como argumentos de entrada el patrón de carga, representado por objetos tipo `LoadPattern` (véase el algoritmo 2.18), el diccionario que relaciona los nodos de la estructura con sus respectivos grados de libertad, calculado con el método `set_indexes()` (véase el algoritmo 2.33), la matriz de rigidez de la estructura, calculada con el método `get_stiffness_matrix()` (véase el algoritmo 2.34), y la matriz de rigidez modificada para tener en cuenta las condiciones de apoyo, calculada con el método `get_stiffness_matrix_with_support()` (véase el algoritmo 2.35).

Según Reddy, 1993, para tener en cuenta las condiciones de apoyo de la estructura en el vector de fuerzas en los nodos, se deben reemplazar los valores asociado a los grados de libertad restringidos por cero.

El vector de fuerzas en los nodos de la estructura del caso de carga se calcula con el método `get_f()` (véase el algoritmo 2.23). Para cada objeto tipo `Support` del diccionario `supports` se extraen los respectivos grados de libertad del diccionario `indexes` y se calculan las restricciones del apoyo con el método `get_restrains()` (véase el algoritmo 2.17), para reemplazar los valores asociados a los grados de libertad restringidos del vector de fuerzas en los nodos de la estructura por cero.

Finalmente, se calculan los vectores de desplazamientos y fuerzas en los nodos de la estructura, y se almacena los resultados en las variables `u` y `f`, respectivamente.

Algoritmo 2.36: Método `solve_load_pattern()` de la clase `Structure`.

```
def solve_load_pattern(self, load_pattern, indexes, k, k_support):
    """
    Solve load pattern

    Parameters
    ----------
    load_pattern : LoadPattern
        Load pattern object.
    indexes : dict
        Key value pairs joints and indexes.
    k : ndarray
        Stiffness matrix of the structure.
    k_support : ndarray
        Stiffness matrix of the structure modified by supports.

    Returns
    -------
    u : ndarray
        Displacements vector.
    f : ndarray
        Forces vector.
    """
    flag_joint_displacements = self.get_flag_active_joint_displacements()

    f = load_pattern.get_f(flag_joint_displacements, indexes).toarray()

    for joint, support in self.supports.items():
        joint_indexes = indexes[joint]
```

```
        restrains = support.get_restrains(flag_joint_displacements)
        for index in joint_indexes[restrains]:
            f[index, 0] = 0

    u = np.linalg.solve(k_support, f)
    f = np.dot(k, u) + load_pattern.get_f_fixed(flag_joint_displacements,
indexes).toarray()

    return u, f
```

2.2.9. set_load_pattern_displacements()

El método `set_load_pattern_displacements()` de la clase `Structure` almacena los desplazamientos de los nodos de la estructura, debidos a las cargas definidas en los patrones de carga, en el diccionario `displacements`.

En el algoritmo 2.37 se presenta la implementación del método `set_load_pattern_displacements()`. El método recibe como argumentos de entrada el patrón de carga, representado por objetos tipo `LoadPattern` (véase el algoritmo 2.18), el diccionario que relaciona los nodos de la estructura con sus respectivos grados de libertad, calculado con el método `set_indexes()` (véase el algoritmo 2.33), y el vector de desplazamientos de los nodos de la estructura, calculado con el método `solve_load_pattern()` (véase el algoritmo 2.36).

Para cada objeto tipo `Joint` del diccionario `joints` se crea una entrada en el diccionario `load_pattern_displacements`, donde las llaves son los objeto tipo `Joint` y los valores objetos tipo `Displacements`, creados con los respectivos valores del vector de desplazamientos de los nodos de la estructura (véase el algoritmo 2.27).

Finalmente, el diccionario `load_pattern_displacements` se almacena en el diccionario `displacements` usando el objeto tipo `LoadPattern` como llave.

Algoritmo 2.37: Método `set_load_pattern_displacements()` de la clase `Structure`.

```
def set_load_pattern_displacements(self, load_pattern, indexes, u):
    """
    Set load pattern displacement

    Parameters
    ----------
    load_pattern : LoadPattern
        Load pattern.
    indexes : dict
        Key value pairs joints and indexes.
```

```
    u : ndarray
        Displacements.
    """
    flag_joint_displacements = self.get_flag_active_joint_displacements()
    load_pattern_displacements = {}

    for joint in self.joints.values():
        joint_indexes = indexes[joint]
        displacements = flag_joint_displacements.astype(float)
        displacements[flag_joint_displacements] = u[joint_indexes, 0]
        load_pattern_displacements[joint] = Displacement(*displacements)

    self.displacements[load_pattern] = load_pattern_displacements
```

2.2.10. set_load_pattern_reactions()

El método `set_load_pattern_reactions()` de la clase `Structure` almacena las reacciones de los apoyos de la estructura, debidos a las cargas definidas en los patrones de carga, en el diccionario `reactions`.

En el algoritmo 2.38 se presenta la implementación del método `set_load_pattern_reactions()`. El método recibe como argumentos de entrada el patrón de carga, representado por objetos tipo `LoadPattern` (véase el algoritmo 2.18), el diccionario que relaciona los nodos de la estructura con sus respectivos grados de libertad, calculado con el método `set_indexes()` (véase el algoritmo 2.33), y el vector de fuerzas en los nodos de la estructura, calculado con el método `solve_load_pattern()` (véase el algoritmo 2.36).

Para cada objeto tipo `Support` del diccionario `supports` se crea una entrada en el diccionario `load_pattern_reactions`, donde las llaves son los objetos tipo `Joint` y los valores objetos tipo `Reactions`, creados con los respectivos valores del vector de fuerzas en los nodos de la estructura (véase el algoritmo 2.28).

Finalmente, el diccionario `load_pattern_reactions` se almacena en el dicionario `reactions` usando el objeto tipo `LoadPattern` como llave.

Algoritmo 2.38: Método `set_load_pattern_reactions()` de la clase `Structure`.

```
def set_load_pattern_reactions(self, load_pattern, indexes, f):
    """
    Set load pattern reactions

    Parameters
    ----------
```

```
    load_pattern : LoadPattern
        Load pattern.
    indexes : dict
        Key value pairs joints and indexes.
    f : ndarray
        Forces.
    """
    flag_joint_displacements = self.get_flag_active_joint_displacements()
    load_pattern_reactions = {}

    for joint in self.supports.keys():
        joint_indexes = indexes[joint]
        reactions = flag_joint_displacements.astype(float)
        reactions[flag_joint_displacements] = f[joint_indexes, 0]
        load_pattern_reactions[joint] = Reaction(*reactions)

    self.reactions[load_pattern] = load_pattern_reactions
```

2.2.11. solve()

El método `solve()` de la clase `Structure` analiza el modelo de la estructura sometida a los diferentes patrones de carga y almacena los resultados en los diccionarios `displacements` y `reactions`.

En el algoritmo 2.39 se presenta la implementación del método `solve()`. El método calcula el diccionario que relaciona los nodos de la estructura con sus respectivos grados de libertad, con el método `set_indexes()` (véase el algoritmo 2.33), la matriz de rigidez de la estructura, con el método `get_stiffness_matrix()` (véase el algoritmo 2.34), y la matriz de rigidez modificada por las condiciones de apoyo, con el método `get_stiffness_matrix_with_support()` (véase el algoritmo 2.35).

Para cada patrón de carga se calculan los vectores de desplazamientos y fuerzas en los nodos de la estructura y los resultados se almacenan en los diccionarios `displacements` y `reactions` respectivamente.

Algoritmo 2.39: Método `solve()` de la clase `Structure`.

```
def solve(self):
    """Solve the structure"""
    indexes = self.set_indexes()

    k = self.get_stiffness_matrix(indexes).toarray()
    k_support = self.get_stiffness_matrix_with_support(k, indexes)

```

```
    for load_pattern in self.load_patterns.values():
        u, f = self.solve_load_pattern(load_pattern, indexes, k, k_support)
        self.set_load_pattern_displacements(load_pattern, indexes, u)
        self.set_load_pattern_reactions(load_pattern, indexes, f)
```

2.2.12. export()

El método `export()` de la clase `Structure` genera un archivo de texto en formato JSON con la descripción del modelo para ser interpretado por el programa de computador *FEM.js*.

El método almacena los objetos que representan los materiales, las secciones transversales, los nodos, los elementos aporticados, las condiciones de apoyo y los patrones de carga, con sus respectivas cargas, en las entradas `materials`, `sections`, `joints`, `frames`, `supports` y `load_patterns`, respectivamente, usando las mismas llaves con las que fueron agregados al modelo.

En el caso donde se usan dichos objetos como llaves para almacenar otros objetos, como es el caso de los objetos tipo `Support` (véase el algoritmo 2.4), o como atributos para crear otros, como es el caso de los objetos tipo `Frame` (véase el algoritmo 2.3), se almacenan las llaves con las que fueron agregados al modelo.

A continuación se presenta la estructura general que tiene un archivo generado por este método.

```
{
    "materials": {
        "key_material": {
            "E": 0.0,
            "G": 0.0
        },
        ...
    },
    "sections": {
        "key_section": {
            "area": 0.0,
            "Ix": 0.0,
            "Iy": 0.0,
            "Iz": 0.0,
            "type": "Section"
        },
        "another_key": {
            "area": 0.0,
            "Ix": 0.0,
```

```
            "Iy": 0.0,
            "Iz": 0.0,
            "type": "RectangularSection",
            width: 0.0,
            height: 0.0
        },
        ...
    },
    "joints": {
        "key": {
            "x": 0.0,
            "y": 0.0,
            "z": 0.0
        },
        ...
    },
    "frames": {
        "key": {
            "j": "key_joint",
            "k": "another_key_joint",
            "material": "key_material",
            "section": "key_section"
        },
        ...
    },
    "supports": {
        "key_joint": {
            "ux": bool,
            "uy": bool,
            "uz": bool,
            "rx": bool,
            "ry": bool,
            "rz": bool
        },
        ...
    },
    "load_patterns": {
        "key_load_pattern": {
            "joints": {
                "key_joint": [
                    {
                        "fx": 0.0,
                        "fy": 0.0,
                        "fz": 0.0,
                        "mx": 0.0,
                        "my": 0.0,
                        "mz": 0.0
                    },
```

```
                    ...
                ],
                ...
            },
            "frames": {
                "distributed": {
                    "local": {
                        "key_frame": {
                            "fx": 0.0,
                            "fy": 0.0,
                            "fz": 0.0
                        },
                        ...
                    }
                }
            }
        }
    }
}
```

2.3. Otras clases

Las clases presentadas hasta aquí permiten analizar linealmente estructuras aporticadas tridimensionales sometidas a cargas estáticas. Adicional a estas clases, en el archivo `classtools.py` se desarolló la clase `AttrDisplay` y la metaclase `UniqueInstances`, las cuales son heredadas por las demás clases.

2.3.1. AttrDisplay

La clase `AttrDisplay` implementa una representación más cómoda de los objetos al redefinir el método `__repr__()`.

En el algoritmo 2.40 se presenta la implementación del método `AttrDisplay`. El método genera una cadena de texto donde aparece el tipo del objeto y entre paréntesis los valores de sus atributos.

Algoritmo 2.40: Clase `AttrDisplay` implementada en el archivo `classtools.py`.

```
class AttrDisplay:
    __slots__ = ()

    def __repr__(self):
        """
```

```
        Get representation object

        Returns
        -------
        str
        Object representation.
        """
        return "{}({})".format(self.__class__.__name__, ', '.join([repr(getattr
    (self, name)) for name in self.__slots__]))
```

2.3.2. UniqueInstances

La metaclase `UniqueInstances` implementa un mecánismo para evitar crear objetos con los mismos atributos de otros objetos de la misma clase, redefiniendo los métodos `__new__()` y `__call__()`.

En el algoritmo 2.41 se presenta la implementación del método `__new__()`. La metaclase redefine la creación de las clases que la implementan, asignándoles un *set*, inicialmente vacío, y *sobrecargando* sus métodos `__setattr__()` y `__del__()`.

En el *set* `instances_attrs` se lleva el registro de los atributos de los objetos existentes de la misma clase, mientras que los métodos `__setattr__()` y `__del__()` actualizan el *set* cuando un atributo de cualquier objeto cambia o cuando el objeto es eliminado, respectivamente.

Algoritmo 2.41: Método `__new__()` de la metaclase `UniqueInstances`.

```
def __new__(mcs, name, bases, dct):
    """
    Create a class

    Parameters
    ----------
    name : str
        Class name.
    bases : tuple
        Parent classes.
    dct : dict
        Namespace's class.
    """
    if '__slots__' in dct:
        dct['instances_attrs'] = set()
        dct['__setattr__'] = UniqueInstances.setattr
        dct['__del__'] = UniqueInstances.delete
```

```
            return type.__new__(mcs, name, bases, dct)
        else:
            print("Warning: " +
                  "Classes created with the UniqueInstances metaclass must implement
    the " +
                  "'__slots__' variable. The class was not created.")
```

En el algoritmo 2.42 se presenta la implementación de la función `setattr`, la cual redefine el método `__setattr__()` de las clases que implementan la metaclase `UniqueInstances`.

Antes que cambie el valor de un atributo de un objeto, este método verifica que los nuevos valores de sus atributos no los tenga ya otro objeto de la misma clase, revisando los elementos almacenados en el *set* `instances_attrs` de la clase.

En caso que no existan objetos con los mismos atributos, se cambia el atributo del objeto y se actualiza el *set* `instances_attrs`. En caso contrario, no se modifica el objeto.

Algoritmo 2.42: Función `setattr` implementada en la clase `UniqueInstances`.

```
def setattr(self, key, value):
    """
    Set attribute object if doesn't collide with attributes another object

    Parameters
    ----------
    key : string
        Key's attribute to modified.
    value : value
        Value to assign.
    """
    if hasattr(self, key):
        # get instances attrs and instance attrs
        instances_attrs = getattr(self.__class__, 'instances_attrs')
        instance_attrs = tuple(getattr(self, name) for name in self.__slots__)

        # get possible new instance attrs
        _instance_attrs = tuple((getattr(self, _key) if _key != key
                                 else value for _key in self.__slots__))

        # add new instance attrs if not in instances attrs
        if _instance_attrs in instances_attrs:
            print("Warning: " +
                  "There is another instance of the class " +
                  "'{}'".format(self.__class__.__name__) +
                  " with the same attributes. The object was not changed.")
```

```
            return None
        else:
            instances_attrs.remove(instance_attrs)
            instances_attrs.add(_instance_attrs)

    self.__class__.__dict__[key].__set__(self, value)
```

En el algoritmo 2.43 se presenta la implementación de la función `delete()`, la cual redefine el método `__del__()` de las clases que implementan la metaclase `UniqueInstances`.

Antes de eliminar todas las referencias a un objeto, este método elimina la entrada asociada del *set* `instances_attrs` de la clase.

Algoritmo 2.43: Function `delete` implementada en la clase `UniqueInstances`.

```
def delete(self):
    getattr(self.__class__, 'instances_attrs').remove(tuple(getattr(self, name
   ) for name in self.__slots__))
```

Finalmente, en el algoritmo 2.44 se presenta la implementación del método `__call__()`. La metaclase evita que se creen objetos con los mismos atributos de otros objetos de la misma clase ya creados, revisando que los atributos del objeto a crear no se encuentren en el *set* `instances_attrs` de la clase.

Algoritmo 2.44: Método `__call__` de la metaclase `UniqueInstances`.

```
    def setattr(self, key, value):
def __call__(cls, *args, **kwargs):
    """
    Return an instances if it does not already exist otherwise return None

    """
    # get __init__ class
    init = cls.__init__

    # get init's arguments and default values
    varnames = getattr(getattr(init, '__code__'), 'co_varnames')[len(args) +
   1:]
    default = getattr(init, '__defaults__')

    # create list with args
    instance_attrs = list(args)

    # fill instance_attrs with kwargs or init's default values
    for i, key in enumerate(varnames):
```

```
        instance_attrs.append(kwargs.get(key, default[i]))

    # from list to tuple
    instance_attrs = tuple(instance_attrs)  # FIXME: i don't need necessary
   check all params

    # get obj's attrs and instances attrs class
    instances_attrs = getattr(cls, 'instances_attrs')

    # check obj's attrs don't be in instances attrs class
    if instance_attrs in instances_attrs:
        print("Warning: " +
                "There is another instance of the class " +
                "'{}' ".format(cls.__name__) +
                "with the same attributes. The object was not created.")
    else:
        # add obj's attrs to instances attrs
        instances_attrs.add(instance_attrs)

        # create and instantiate the object
        obj = cls.__new__(cls, *args, **kwargs)
        obj.__init__(*args, **kwargs)

        return obj
```

Bibliografía

Akademiia nauk SSSR. (1763). *Novi comementarii Academiae scientiarum imperialis petropolitanae*. Typis Academiae Scientarum.

Chacon, S. (2014). *Pro Git*. Berkeley, CA New York, NY, Apress,Distributed to the Book trade worldwide by Spring Science+Business Media.

Computers & Structures. (2017). *CSi Anlysis Reference Manual*.

Computers & Structures. (2019). *Welcome to ETABS*.

Computers & Structures. (2020). ETABS System Requirements [Accedido: 2020-09-29].

Dirksen, J. (2015). *Learning Three.js–the JavaScript 3D library for WebGL : create stunning 3D graphics in your browser using the Three.js JavaScript library*. Birmingham, UK, Packt Publishing.

Dunn, F. (2002). *3D math primer for graphics and game development*. Plano, Tex, Wordware Pub.

Escamilla, J. (1995). *Microcomputadores en ingeniería estructural*. Santafé de Bogotá, ECOE Universidad Nacional de Colombia. Facultad de Ingeniera.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del R'ıo, J. F., Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357-362. https://doi.org/10.1038/s41586-020-2649-2

Lutz, M. (2013). *Learning Python*. Sebastopol, CA, O'Reilly.

Reddy, J. N. (1993). *An introduction to the finite element method*. New York, McGraw-Hill.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261-272. https://doi.org/10.1038/s41592-019-0686-2

Weaver, W. J. & Gere, J. (1990). *Matrix analysis of framed Structures*. New York, Van Nostrand Reinhold.

Wilson, E. L. & Dovey, H. H. (1972). Three dimensional analysis of building systems - TABS. *Earthwuake engineering research center*.

Wilson, E. L., Hollings, J. P. & Dovey, H. (1975). Three dimensional analysis of building systems (extended version). *Earthwuake engineering research center*.